ACCURATE AND INTERACTIVE VISUALIZATION OF HIGH-ORDER FINITE ELEMENT FIELDS

by

Blake William Nelson

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computing

School of Computing

The University of Utah

August 2012

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Blake William Nelson**

has been approved by the following supervisory committee members:

Robert M. Kirby	, Chair	**4/13/2012** Date Approved
Robert Haimes	, Member	**4/13/2012** Date Approved
Christopher Johnson	, Member	**4/13/2012** Date Approved
Steven Parker	, Member	**4/13/2012** Date Approved
Peter Shirley	, Member	**4/13/2012** Date Approved

and by **Alan Davis** , Chair of the Department of **Computer Science**

and by Charles A. Wight, Dean of The Graduate School.

ABSTRACT

High-order finite element methods, using either the continuous or discontinuous Galerkin formulation, are becoming more popular in fields such as fluid mechanics, solid mechanics and computational electromagnetics. While the use of these methods is becoming increasingly common, there has not been a corresponding increase in the availability and use of visualization methods and software that are capable of displaying visualizations of these volumes both accurately and interactively. A fundamental problem with the majority of existing visualization techniques is that they do not understand nor respect the structure of a high-order field, leading to visualization error. Visualizations of high-order fields are generally created by first approximating the field with low-order primitives and then generating the visualization using traditional methods based on linear interpolation. The approximation step introduces error into the visualization pipeline, which requires the user to balance the competing goals of image quality, interactivity and resource consumption. In practice, visualizations performed this way are often either undersampled, leading to visualization error, or oversampled, leading to unnecessary computational effort and resource consumption.

Without an understanding of the sources of error, the simulation scientist is unable to determine if artifacts in the image are due to visualization error, insufficient mesh resolution, or a failure in the underlying simulation. This uncertainty makes it difficult for the scientists to make judgments based on the visualization, as judgments made on the assumption that artifacts are a result of visualization error when they are actually a more fundamental problem can lead to poor decision-making.

This dissertation presents new visualization algorithms that use the high-order data in its native state, using the knowledge of the structure and mathematical properties of these fields to create accurate images interactively, while avoiding the error introduced by representing the fields with low-order approximations. First, a new algorithm for cut-surfaces is presented, specifically the accurate depiction of color-

maps and contour lines on arbitrarily complex cut-surfaces. Second, a mathematical analysis of the evaluation of the volume rendering integral through a high-order field is presented, as well as an algorithm that uses this analysis to create accurate volume renderings. Finally, a new software system, the Element Visualizer (ElVis), is presented, which combines the ideas and algorithms created in this dissertation in a single software package that can be used by simulation scientists to create accurate visualizations. This system was developed and tested with the assistance of the ProjectX simulation team. The utility of our algorithms and visualization system are then demonstrated with examples from several high-order fluid flow simulations.

CONTENTS

ACKNOWLEDGMENTS

I would like to thank Tiago Etiene and Mathias Schott for useful discussions about volume rendering and quadrature, and Sergey Yakovlev for reviewing preliminary drafts of portions of this work and providing needed feedback and direction. I would also like to thank the entire ProjectX team, and specifically Eric Liu, for their many contributions, including the solutions used in the examples used and many insightful discussions over the course of this work.

My appreciation is also extended to Pete, Steve and Chris for comprising a valuable part of my committee, as well as Bob, who went above and beyond the requirements of a committee member. Finally, my thanks to Mike who continually guided me in my research efforts.

This work is supported under ARO W911NF-08-1-0517 (Program Manager Dr. Mike Coyle), Department of Energy (DOE NET DE-EE0004449). Infrastructure support provided through NSF-IIS-0751152.

CHAPTER 1

INTRODUCTION

High-order finite element methods (a variant of which are the spectral/hp element methods considered in this work [24]) have reached a level of sophistication that they are now commonly applied to many real-world engineering problems, such as those found in fluid mechanics, solid mechanics and electromagnetics [46, 23]. These types of simulations are characterized by the use of high-order (i.e., nonlinear) basis functions to represent the simulation mesh and resulting fields. Some commonly used basis functions include nonlinear polynomials, trigonometric functions, wavelets and splines.

Over the last 15 years, an increasing emphasis has been placed on providing visualization algorithms and corresponding software solutions tailored specifically to high-order methods. Despite this, however, there are still few high-order visualization algorithms in existence, and they exist solely as research papers and prototype software that is not easily available for arbitrary simulation packages and data representations. As a result, high-order solutions are still typically visualized using well-established algorithms based on linear primitives rather than using the newer, high-order methods.

In order to be visualized with linear methods, a high-order solution must first be converted into a compatible format. Some common approaches are: sample onto a regular grid of points, using trilinear interpolation between points; convert high-order elements to linear elements by ignoring each element's high-order nodes; or, in the case of surfaces, represent the high-order surface with a polygonal mesh. Each of these approaches approximate the high-order data of interest with a low-order representation. The low-order approximation allows linear algorithms to perform visualizations of high-order data, but this capability comes at a cost; the linear

approximations used to represent the high-order data introduce error into the final image and are computationally expensive.

Despite the error introduced by linear approximation, these approaches are still frequently used for high-order visualization. The reasons for this are compelling: first, there exists an extensive collection of visualization techniques that expect linear primitives as input; and second, linear methods can be rendered interactively on standard desktop computers, while existing high-order methods are often either too slow to be practical, have expensive hardware requirements, or, as noted above, are not available for the basis functions to be visualized. However, as we can see in Figure 1.1, linear methods can not only introduce error into the visualization, they can introduce significant error that is hard to detect visually. In this figure we show a color-mapped cut-plane with contour lines. On the left are the contours generated using VTK [52] (which uses linear interpolation across triangle edges to determine the contour's location), and on the right are the contours generated by our system. The key to this comparison is that the visual cues most often used to determine if an image contains error are either not present (there is no color smearing in the left image) or are actually features of the accurate image (sharp corners in the contour lines on the right).

This leads to the common question: are the features and artifacts seen in the visualization a feature of the high-order data, or are they errors introduced through the visualization process? Unfortunately, visualization errors arising from linear interpolation often look the same as genuine solution artifacts arising from problems such as insufficient mesh resolution. With traditional interpolation-based rendering techniques, engineers are hard-pressed to differentiate between the numerous potential causes of visual artifacts. The severity of these visualization errors can be mitigated by refining the linear approximations, but this approach does not scale well, requiring too much computational time and storage to be practical [51]. While there are techniques for addressing the sampling problem [51, 31, 47], it is often desirable to skip the intermediate step and visualize the high-order data directly (i.e., in its native form) in order to avoid the associated approximation errors. By using the high-order data directly, we can know that any features present in the visualization are also present

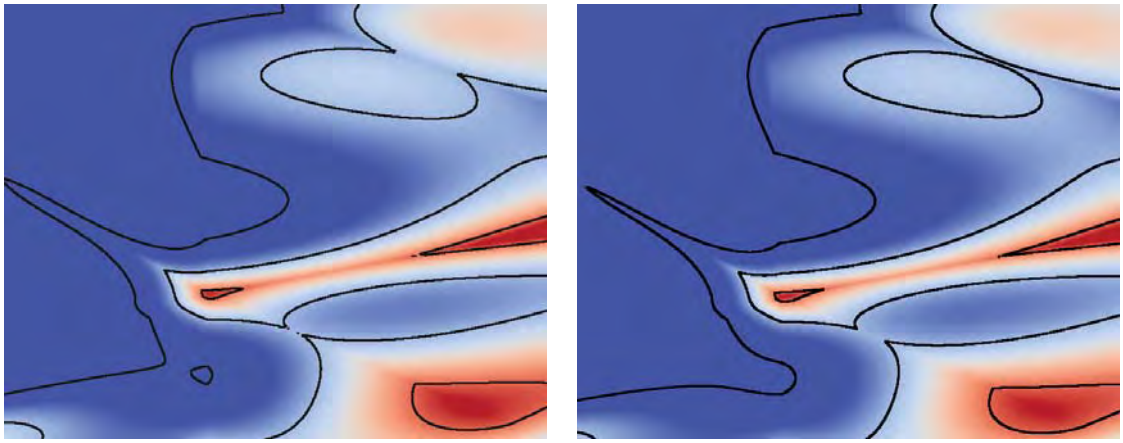


Figure 1.1. Illustration of the difficulty involved in detecting error visually: (Left) Contour lines on a cut-plane through a high-order finite element simulation, rendered using low-order methods. There are no jagged lines or other visual indicators of error. (Right) The accurate contours of the same cut-plane rendered using our algorithms. There is considerable error that is not readily apparent without access to the true solution.

in the data and are not artifacts of the approximation.

1.1 Contributions

This dissertation seeks to create a system for high-order finite element visualization that is both *accurate* and *interactive*. Accuracy is essential for the correct interpretation of visualization results, as well as for debugging simulation codes (examples of which are shown in Chapter 7). Interactivity is critical, as slow visualization software will not be used, as evidenced by the continued use of linear methods to visualize high-order fields, even though more accurate methods exist. In this work, we seek to balance these competing goals, but fall on the side of accuracy when a tradeoff must be made.

The following contributions have been provided in the course of meeting this goal:

- *An algorithm for the visualization of cut-surfaces through high-order fields.* A surface visualization is one in which a scalar field is plotted on a surface using color maps, isocontours or both. These types of visualizations, while conceptually simple, are very useful for the engineer when studying the simulation. Much of the interesting behavior occurs on or near certain domain boundaries

(e.g., a wing), and it makes sense to be able to plot the behavior of the field on these surfaces accurately.

- *An analysis of volume rendering high-order fields.* It is natural to assume that volume rendering of high-order fields can be accelerated through the use of high-order quadrature techniques to evaluate the volume rendering integral. We develop a volume rendering algorithm using the properties of spectral/*hp* elements to categorize each ray based upon the properties of the transfer function composed with the scalar field. We prove that, in the optimal case of fields that are smooth along the ray, the evaluation of the volume rendering integral will generally exhibit second-order convergence, with a worst-case of first-order convergence. We use these properties to develop an optimized high-order volume rendering algorithm in which we first categorize each ray based on the local properties of the transfer function and scalar field and then evaluate the volume-rendering integral with a quadrature method optimized for the category.

- *An implementation of these methods in a visualization system.* There is a need among simulation scientists to have a single visualization package that offers common visualization techniques optimized for their high-order fields. There are a variety of algorithms and prototype applications being developed for high-order data, but there has not been an effort to create an application to combine and unify them into a cohesive system. Until this is done, users will continue to use existing, low-order visualization systems, not because they do not want accuracy, but they need a system that works, and can deal with the accuracy issue in an ad-hoc fashion. We address this issue with the development of a high-order visualization system, the Element Visualizer (ElVis), which is designed to be a comprehensive visualization system for high-order data. This system was developed in close cooperation with the ProjectX simulation team from the Department of Aeronautics and Astronautics at MIT, both to ensure required features are implemented, and to verify the effectiveness of the provided visualization algorithms.

CHAPTER 2

PREVIOUS WORK

In this chapter, we document the existing algorithms designed for the direct visualization of high-order fields

2.1 Linear Methods

To avoid the computational and storage penalties associated with the straightforward linear tessellations mentioned in the introduction, several adaptive subdivision schemes have been developed [47, 50] to reduce the number of linear primitives required. Subdivision is guided through the use of error estimates, which results in few subdivisions for elements well approximated by linear approximation, and many subdivisions for elements that are not. Adaptive subdivision does not eliminate the error introduced into the simulation pipeline, and reducing the error to arbitrary levels can still produce unacceptably large data sets and long execution times.

2.2 Color Maps on Cut-Surfaces

As we will discuss in Chapter 5, the display of cut-surfaces is an effective visualization method for scalar fields. While there are no existing algorithms that apply color maps to arbitrary cut-surfaces directly, there are several schemes that apply color maps to cut-planes. In one approach, the color map is generated by what is called a *polynomial basis texture* [17], in which a texture map corresponding to each unique basis function used in the simulation is created by sampling the basis function in reference space, then using that sample to create the texture map. When rendering a cut-plane, each triangle on the surface is assigned a linear combination of one or more of these basis textures, depending on the element's order. In this way, a set of basis textures can be generated in a preprocessing step, and then, assuming there is sufficient resolution in the texture, accurate images can be generated for all high-order

triangles. Accuracy is closely tied to the size of the basis textures. If the basis textures are not large enough then the image can become inaccurate from some viewpoints. This algorithm is also limited to linear color maps since the linear combination of basis textures will give the wrong result if the color map is not linear.

Another method, is an OpenGL fragment shader that calculates the scalar value at each pixel on a cut plane, resulting in a more accurate lookup into the color map [5]. This is an improvement over the previous implementation because it can calculate the correct color for each pixel in a view independent manner, and it can also handle nonlinear color maps. It is, however, limited to planar surfaces.

Finally, another method analytically calculates the intersection of a plane and quadratic tetrahedra, then uses a ray tracer to apply the color map to the new primitive [62].

2.3 Contours on Cut-Surfaces

Most of the work that generates contour lines deals only with 2D high-order elements. A common theme is to generate the contours in an element's reference space (which we will define in Section 3.1) and then transform them into global (world) space for display. One approach [35] creates contour lines in an element's reference space by subdividing the domain and using linear interpolation within these subdomains to create a piecewise-linear contour. Another approach steps along a direction orthogonal to the field's gradient [4], where each step is controlled by a user-defined step size. A method for generating contour lines over quadrilateral elements by determining the shape of the contour in reference space and then generating a linear polyline to approximate it was developed [53] and later extended to linear and quadratic triangles [54].

The only 3D contouring algorithm [28] generates contour lines on cut-planes through finite element volumes. The procedure first locates a seed point for the contour line along the element's boundary. It then steps in a direction orthogonal to the field's gradient, using a user-controlled step size, to generate a polyline representing the contour. It differs from the previously described methods in that the plane is a 3D entity. At each step, the contour can, and often does, move off of the cut-surface.

The method introduces a correction term to fix this problem and keep the contour on the cut-plane. As with the other object-space contour methods described, the step size is useful to determine how accurately the polyline represents the contour in world space, but is not as useful in expressing how accurate the final image is, as it can be accurate from one view but have large error in another.

Approaches that use interval methods for generating contours [55] assume that, once a suitable region has been isolated, that the intersection between the implicit and a line segment can be computed efficiently.

2.4 Volume Rendering and Isosurfaces

While direct volume rendering of low-order data is treated extensively in the literature [33, 32, 40], there are relatively few methods designed specifically for high-order data. In this section, we provide a brief overview of existing high-order visualization methods.

Since the solution to the volume rendering equation does not, in general, have an analytic solution, it is not sufficient to use the high-order data directly to guarantee accuracy. This is because the volume rendering integral will be approximated with numerical quadrature, which will introduce error into the visualization. An exception to this is when rendering linear tetrahedra [64], for which the volume rendering integral can be solved analytically. While highly accurate results can be obtained for quadratic tetrahedra [62] and hexahedra [64], other element types and polynomial orders cannot be solved analytically and require numerical quadrature. More recently, direct volume rendering of arbitrary elements and polynomial order was introduced [59], where clusters of GPUs are able to produce interactive results. In the special case of volume rendering fields with no transfer function and no emissive component, high-order quadrature techniques can be used to evaluate the attenuation portion of the volume rendering integral without error, resulting in pixel-exact images [48]. Another approach uses point-based samples to approximate the volume rendering integral [69], but sacrifices accuracy for execution speed.

Higher-order integration methods have been investigated in an attempt to achieve faster convergence of the volume rendering integral and to provide better error bounds

on the resulting calculation. An investigation of Simpson's rule applied to voxel-based data [11] showed improved accuracy at the expense of increased memory usage in the preintegration table. Conservative bounds on the number of samples required for a given level of accuracy has been developed for trapezoidal and Simpson's rule, as well as for a power series approach [43].

Approaches for isosurface rendering have been developed for quadratic tetrahedra using analytical calculation of the isosurface in reference space [63] and through ray-tracing approaches [62]. Other approaches include using a ray tracer for arbitrary elements of arbitrary order [42] and a point-based approach that uses particles that actively seek and distribute themselves on the isosurface [36].

2.5 Vector Fields

While most existing work has focused on the visualization of scalar fields, some work has been made to address vector fields as well. Streamlines are a popular method for vector field visualization since they effectively convey the salient features of the field. Generating streamlines of high-order vector fields presents unique challenges since the integration techniques used to create them typically have assumptions about the smoothness of the field that are violated by the high-order field at the element boundaries. Continuous Galerkin (CG) fields do not have continuous derivatives at element boundaries. This means that streamline generation must be done in a carefully controlled manner so that the integration accurately accounts for the discontinuities at element faces [7]. Streamlines of discontinuous Galerkin (DG) fields are discontinuous, which is not a generally expected feature of streamlines. One way to address this problem is to apply a filter to the DG field in a post-processing step that restores continuity to the field while maintaining the solution's accuracy [56, 61].

CHAPTER 3

BACKGROUND

This chapter provides an overview of high-order finite element methods, focusing specifically on those features that are relevant for visualization.

3.1 Finite Elements

The purpose of this section is to provide a high-level overview of the key features of spectral/hp elements used in this work. An in-depth discussion of the finite element method is beyond the scope of this chapter. We refer the interested reader to the following reference works for a more in-depth discussion of finite elements and, in particular, spectral/hp (high-order finite) elements [22, 57, 24, 9]. Instead, we will focus our discussion on those features of spectral/hp methods that are relevant to visualization.

One important property of finite element solution methods is the declaration of the space of admissible solutions. Quite often the first step in the finite element methodology is to define, for the domain Ω over which the partial differential equation(s) of interest are being solved, a tessellation $\mathcal{T}(\Omega)$ of Ω. Four basic element types often used in the construction of the tessellation $\mathcal{T}(\Omega)$ are the hexahedron, tetrahedron, prism, and pyramid, as presented in Figure 3.1.

The result of a finite element simulation is a scalar field $F(\vec{x}) : \Omega \rightarrow \mathbb{R}$ about which certain properties are known. This field will be referred to as the high-order field. In particular, when dealing with continuous Galerkin formulations (CG) we know the field over Ω is a C^0 function where a discontinuity in the derivative occurs at element boundaries. When dealing with discontinuous Galerkin formulations (DG), there are no continuity requirements between elements. In both CG and DG, the field is C^p on the interior of each element, with $p \geq 1$ indicating higher levels of smoothness depending on the element's approximating polynomial order and its mapping to world

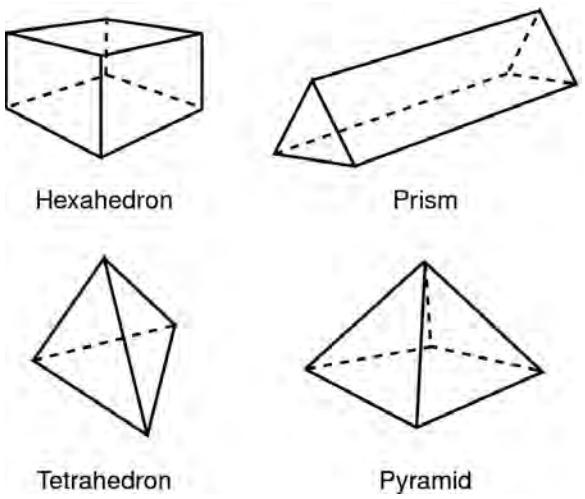


Figure 3.1. Schematic of the four basic element types in three dimensions: hexahedron, prism, tetrahedron and pyramid.

space. The premise of this work is that since finite elements provide functions with explicit levels of smoothness, this structure should be both *respected* and *exploited* in the visualization.

To obtain greater accuracy, either the mesh can be refined by subdividing existing elements into smaller elements (h refinement), or the approximating polynomial order in an element can be increased (p refinement). Deciding whether to refine the element size, order, or both is a nontrivial task [60].

3.1.1 Element Mappings

The solution is defined as a polynomial function on the reference element $F(\xi) \in \mathcal{P}^{N_1,N_2,N_3}$ where N_1, N_2, N_3 denote (possibly) different polynomial orders in the three principle directions. A mapping function Φ is defined for each element that transforms points from reference space to world space. An illustration of this mapping is shown in Figure 3.2, which shows a reference hexahedron and the mapping to the element's actual position and orientation in world space. Since the solution is not defined with respect to world space coordinates, a query for the solution value at point $\vec{x}$ is found by evaluating $F(\Phi^{-1}(\vec{x}))$. In the finite element packages considered in this work, the inverse mapping Φ^{-1} is, in general, not known analytically and must be calculated numerically when needed.

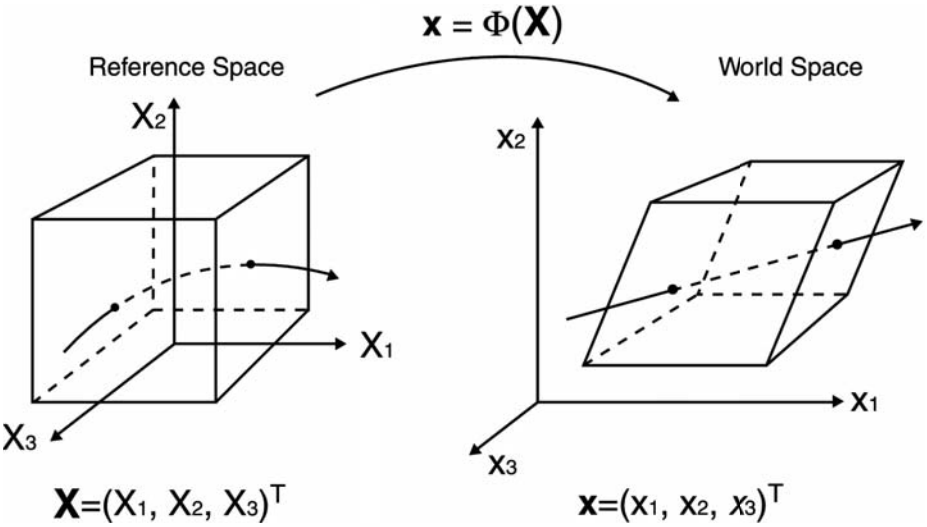


Figure 3.2. Illustration of the mapping between world and reference space for a hexahedron.

Due to the nature of the mapping function $\mathbf{x} = \Phi(\mathbf{X})$ used in spectral/*hp* element methods (for instance, in the case of a hexahedron, a trilinear mapping from reference space to world space is used [24]), a line segment in world space does not always correspond to a line segment in reference space. On the contrary, it is often the case that the mappings cause line segments in world space to be curves in reference space (as illustrated in the schematic given in Figure 3.2). Hence one is often forced to decide between working with a polynomial function in reference space along a not-necessarily-parameterizable curve, or a smooth function along a line segment in world space. In this work, we choose the latter approach.

Although spectral/*hp* element methods [24] seek polynomial solutions *with respect to the reference element*, the solution with respect to world space coordinates may not be polynomial (and in general is not). Under the mappings used in spectral/*hp* finite elements, we are guaranteed that the solution is at least C^0 continuous and smooth on the interior of each element. In this work, we assume that all discretizations investigated respect smoothness criteria on the interior of each element. We exploit the smoothness of the function within each element in each of algorithms described in the following chapters.

3.1.2 Element Field Evaluation

After finding the intersection point $\boldsymbol{x}$ and the element $\boldsymbol{E}$ that contains the point, the field can be evaluated. The field is defined in terms of the local Cartesian coordinate system associated with $\boldsymbol{E}$'s tensor element. As we discussed in Section 3.1, the mapping between the local tensor space and the global Cartesian coordinate system (in which $\boldsymbol{x}$ exists) is specified by the function $\Phi_e(\boldsymbol{\xi})$. Because Φ_e is a bijection *a.e.*, we can obtain the tensor space point for a given world space point by inversion. To calculate the field's value at a world point $\boldsymbol{x}$ inside the element:

$$\hat{F}(\boldsymbol{x}) = F(\Phi_e^{-1}(\boldsymbol{x})). \tag{3.1}$$

Unfortunately, Φ_e does not, in general, have an analytic inverse. We therefore perform the inverse mapping numerically. We use the Newton-Raphson algorithm, which generates the convergent sequence

$$\boldsymbol{\xi}_{i+1} = \boldsymbol{\xi}_i + \boldsymbol{J}^{-1}\left(\Phi(\boldsymbol{\xi}_i) - \boldsymbol{x}\right) \tag{3.2}$$

where $\boldsymbol{J}$ is the Jacobian of the mapping function given by

$$J_{ij} = \frac{\partial \Phi_i}{\partial \xi_j} \tag{3.3}$$

with Φ_i as the mapping in the i direction. This method has well-known stability issues and can fail to converge, even when roots do exist. In our case, since the mapping function Φ is a bijection *a.e.*, we know that there is a unique real root inside the element, and we are able to position our initial guess close to the root, which greatly increases the probability of finding the root. Our routines do detect and report convergence errors and, in practice, they have not been encountered.

Fields are represented as the tensor product of one-dimensional polynomials:

$$F(\xi_1, \xi_2, \xi_3) = \sum_i \sum_j \sum_k \hat{u}_{ijk} \phi_i(\xi_1) \phi_j(\xi_2) \phi_k(\xi_3). \tag{3.4}$$

Implemented as written, if each polynomial is of order N, evaluating the field is an $O(N^4)$ operation ($O(N)$ to evaluate the polynomials in each iteration, and $O(N^3)$ iterations), but if we use the sum factorization technique

$$F(\xi_1, \xi_2, \xi_3) = \sum_i \phi_i(\xi_1) \sum_j \phi_j(\xi_2) \sum_k \hat{u}_{ijk} \phi_k(\xi_3) \tag{3.5}$$

this is reduced to $O(N^2)$. Due to the number of samples that must be taken to provide an accurate visualization, using the sum factorization technique can provide significant performance gains, especially as the field's polynomial order becomes large.

3.2 Interval Arithmetic

Interval arithmetic was introduced as a way to bound the variation in function evaluation [39].

Using interval arithmetic, we replace operations on real numbers with operations on intervals. An interval X is defined as

$$X = [\underline{X}, \overline{X}] = \{x \in \mathbb{R} : \underline{X} \leq x \leq \overline{X}\} \tag{3.6}$$

and, for an arbitrary function $\oplus$ and intervals X and Y:

$$X \oplus Y = \{x \oplus y : x \in X \wedge y \in Y\}. \tag{3.7}$$

The set image of a function g is defined as:

$$g(X) = \{g(x) : x \in X\} \tag{3.8}$$

and represents the true range of g over the interval X. The interval extension G of g is formed by evaluating the steps to calculate g on interval numbers rather than floating point numbers. The interval extension has the following useful property:

$$G(X) \supseteq g(X) = \{g(x, y, z) : (x, y, z) \in X\}. \tag{3.9}$$

In other words, if g is evaluated using interval arithmetic on an input interval X, the result is a range that is guaranteed to contain the true range of g on that interval.

The interval extension G of g can be very wide, and in some cases can be too wide to be useful. To reduce the width of the interval, we first subdivide the interval into n segments of width h, such that the subdivision of X is:

$$X_i = [\underline{X} + ih, \underline{X} + (i+1)h]. \tag{3.10}$$

We define the interval hull of two intervals as:

$$X \underline{\cup} Y = [min\{\underline{X}, \underline{Y}\}, max\{\overline{X}, \overline{Y}\}]. \tag{3.11}$$

The interval hull of a subdivided interval is

$$G_n = \underline{\bigcup}_{i \leq n} X_i. \tag{3.12}$$

Subdividing an interval is useful because of the following property:

$$G(X) \supseteq G_n(X) \supseteq g(X) = \{g(x, y, z) : (x, y, z) \in X\}. \tag{3.13}$$

In other words, we can decrease the width of an interval computation by subdividing the interval into smaller regions, evaluating each of the subintervals separately, then calculating the interval hull of the smaller intervals (see Figure 3.3).

To illustrate how this works, consider the the polynomial $3x^2 + 2x - 4$ which has a range of $[-4.33, 1]$ over $[-1, 1]$. Using the following properties of interval arithmetic:

$$\begin{aligned} X + Y &= [\underline{X} + \underline{Y}, \overline{X} + \overline{Y}], \\ X - Y &= [\underline{X} - \overline{Y}, \overline{X} - \underline{Y}], \end{aligned}$$

$$X^n = \begin{cases} [\underline{X}^n, \overline{X}^n] & \text{if } \underline{X} > 0 \text{ or } n \text{ is odd}, \\ [\overline{X}^n, \underline{X}^n] & \text{if } \overline{X} < 0 \text{ or } n \text{ is even}, \\ [0, |X|^n] & \text{if } 0 \in X \text{ and } n \text{ is even}, \end{cases}$$

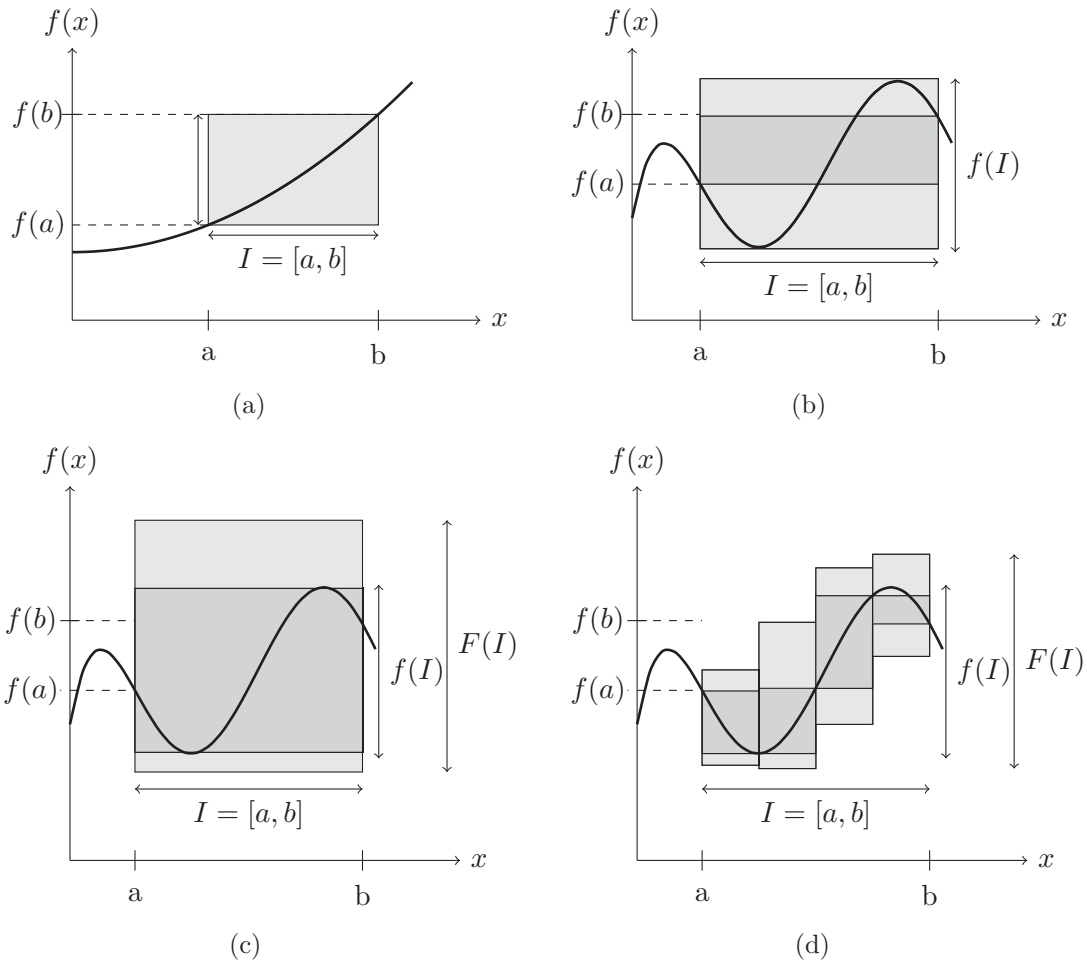


Figure 3.3. Using interval arithmetic to estimate the range of a function. (a) If function f is monotonic on an interval $I = [a, b]$, then the range of f is bounded by $f(I) = [f(a), f(b)]$. (b) If f is not monotonic, then $f(I) \neq [f(a), f(b)]$. (c) An interval extension F of f provides bounds that include $f(I)$, i.e., $f(I) \subseteq F(I)$. (d) Uniform subdivision of the domain produces tighter bounds.

the range can be evaluated as $3[-1, 1]^2+2[-1, 1]-4 = 3[0, 1]+[-2, 2]-[4, 4] = [-6, 1]$. Note the true range, $[-4.33, 1] \subseteq [-6, 1]$. Dividing $[-1, 1]$ into 10 evenly-spaced intervals and evaluating produces a range of $[-4.72, 1]$ and 100 evenly-spaced intervals produces $[-4.37, 1]$.

3.3 Ray-Tracing

Ray-tracing is a viewing paradigm characterized by a viewing ray cast from an eyepoint through an image plane. The behavior of the ray as it interacts with the scene

determines the final color of each pixel. Different methods are often characterized as either *ray-casting* or *ray-tracing*. The former is often applied to methods consisting of rays that are cast from the eyepoint and terminate at their first intersection, while the latter method is often characterized by additional rays cast from the initial intersection point to produce effects such as lighting and shadows.

Rays are generally represented in parametric form:

$$\boldsymbol{r} = \vec{o} + t\vec{d} \tag{3.14}$$

where $\vec{o}$ is the ray's origin, $\vec{d}$ is the ray's direction, and t is the distance along the ray from the origin along the specified direction. A typical ray-tracing scenario consists of casting rays of this form into the scene, then testing each geometric object in the scene for an intersection.

Implicit surfaces are defined as $f : \mathbb{R} \to \mathbb{R}, f(x, y, z) = 0$, for which substitution produces an equation

$$f(o_x + d_x t, o_y + d_y t, o_z + d_z t) = 0. \tag{3.15}$$

Solving this equation is a one-dimensional root finding problem. Finding these roots is equivalent to finding all of the ray-object intersections along the ray.

Another common surface type is a parametric surface:

$$\vec{x} = f(r, s). \tag{3.16}$$

Simple substitution does not work in this scenario, so instead we solve the following system of equations:

$$\begin{aligned} f_x(r, s) - o_x - d_x t &= 0 \\ f_y(r, s) - o_y - d_y t &= 0 \\ f_z(r, s) - o_z - d_z t &= 0. \end{aligned} \tag{3.17}$$

This type of surface appears frequently in the context of high-order finite element volumes as the faces of curved elements, or the curved geometry that is part of the simulation.

3.4 Visualization Verification

Visualization is the lens through which scientists and engineers interpret their simulation results and view modeling and discretization interactions (see Chapter 7 for examples). Given that visualization occupies an important place in the overall interpretation of a simulation's result, it is surprising that there has not been much consideration to date of the errors involved. In this section, we give a brief overview of the *error budget* of the simulation pipeline, and then show how that relates to high-order simulations and their visualization.

In Figure 3.4 (from [25] with the authors' permission), we show the simulation pipeline which is commonly used by scientists and engineers. The process starts with a problem of interest and a collection of questions about the process that needs answers. The next stage is the development of a mathematical model of the physical process, which abstracts the salient features of the problem so that the resulting problem is tractable and the model answers the desired questions. This is followed by an implementation of the mathematical problem as a computer program, which is then executed to generated the desired solutions. Visualization is then performed on the output of the simulation.

Each stage of this process introduces error. Validation is the process of determining whether the mathematical model represents the physical process with sufficient accuracy. Verification is the process of determining if the code written to implement

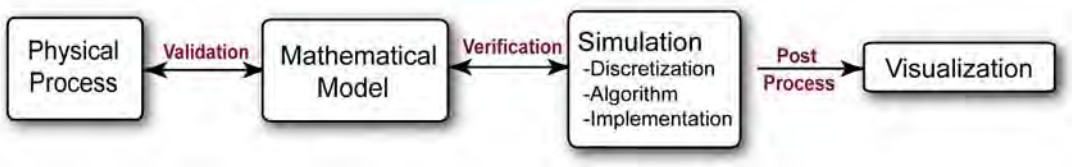


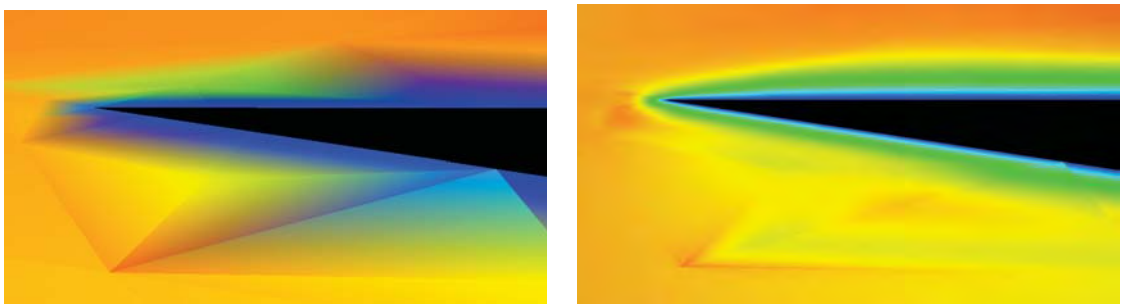
Figure 3.4. Overview of the simulation pipeline, from the phenomenon of interest to the final image (from [25] with the authors' permission).

the mathematical model can calculate the result to a specified level of accuracy. Verification includes such things as analyzing discretization error and verifying that the code is written correctly.

Visualization is often performed in an ad-hoc fashion, with the commonly held belief that the visualization can have error and still convey the essential points of the simulation's results. There is often an implicit, assumption that error in the resulting visualization will be obvious and can be addressed by increasing the visualization discretization. Unfortunately, when visualized using existing techniques based on linear interpolation, it is often unclear if the resulting image conveys an accurate representation of the underlying data, leading to the common question: are the features and artifacts seen in the visualization part of the high-order data, or are they errors introduced through the visualization process? Unfortunately, visualization errors arising from linear interpolation often look the same as genuine solution artifacts arising from problems such as insufficient mesh resolution (see Figure 3.5). The severity of these visualization errors can be mitigated by refining the linear approximations, but this approach does not scale well, requiring too much computational time and storage to be practical [51]. With traditional interpolation-based rendering techniques, engineers are hard-pressed to differentiate between the numerous potential causes of visual artifacts.

What is needed is to apply the principles of validation and verification to visualization, which has recently gained renewed attention [25, 13, 14]. The revised pipeline is shown in Figure 3.6. In this pipeline, we address the way in which we represent the data (validation) and we verify the result of our visualizations (verification).

For the case of linear representations, error can be reduced by subsampling and/or creating a finer linear representation of the underlying data. This can be done either by brute force (such bisecting all linear segments in the approximation), or adaptively, through the use of error metrics, where we refine the approximations until the resulting error is less than some predetermined amount. There are several drawbacks to this approach. First, it is computationally expensive. The time and resource usage of brute force approaches grow exponentially, and are not useful for practical visualization. Adaptive methods can provide better results, but still end up



(a) (b)

Figure 3.5. An illustration of the difficulty of distinguishing visualization and simulation artifacts. (a) - Visualization of a cut-plane through a fluid-flow field in which there are several obvious artifacts. (b) - The same view using our visualization system. We can see that some artifacts disappear in the accurate image, implying they are caused by the visualization algorithm, while others remain, implying they are part of the underlying data model.

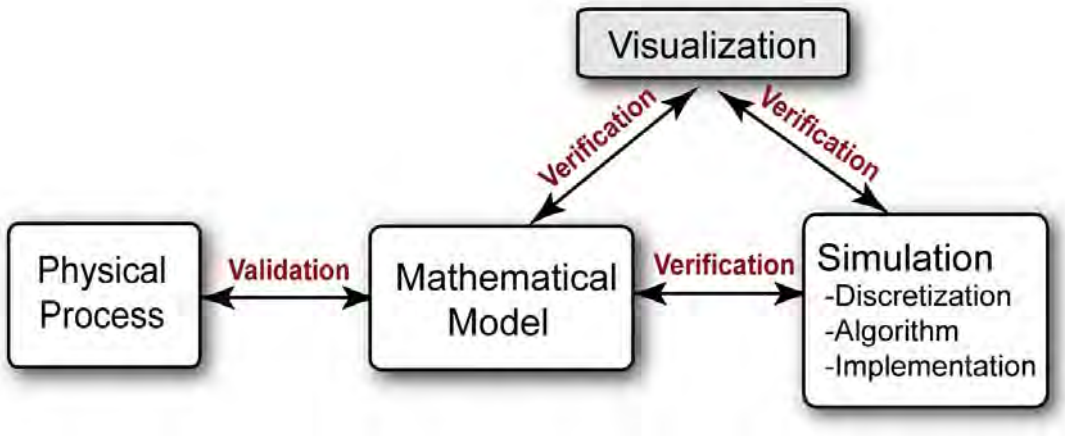


Figure 3.6. Revised simulation pipeline with validation and verification included in the visualization (from [25] with the authors' permission).

using significant amounts of time and memory. And second, the adaptation is not scale-invariant. In other words, sampling the volume to a particular level of error may be acceptable for visualizations that span the entire volume, but be entirely inadequate for closeup views of local regions of interest. Errors that scale with the view will require runtime resampling of the high-order data set at runtime.

The approach we advocate is to use the data in its native form. In this way, there is no need to perform validation as we are not using a different representation of the

data; we have no representation error in our visualization pipeline. The downside is that this approach means we can't use the majority of existing visualization packages and code, as they will be unable to display our data. Another disadvantage is that evaluating the high-order field for visualization requires significantly more floating point operations than are required to evaluate linear fields. Therefore, it takes more effort and algorithmic manipulations to create interactive visualizations.

With this background, we can now define a term that will be used frequently throughout the rest of this work: *pixel-exact images*. At a high level, what we mean by a pixel-exact image is one in which there is no error; a viewer of the image can have confidence that the image being presented is, in some sense, an accurate representation of the underlying data. More formally, a pixel-exact image is one in which the error introduced through the combination of visualization modeling error and visualization algorithm discretization error is less than 0.001 for each color channel in the resulting image. We choose 0.001 as our cutoff because, for an 8-bit per channel color image, a pixel needs an error of $1/256 = 0.00390625$ to be a different color than the true color. By using a slightly more strict definition, we guard against round-off errors in algorithms that try to adaptively control the error. This definition of image accuracy deals only with those errors introduced after the simulation has been completed. Errors in mathematical modeling and simulation may still exist and will appear in a pixel-exact image (for an example of how this can be used for debugging a simulation, see Section 7.4).

In practice we do not, in general, know if our algorithm is producing images with this amount of error; if we knew the exact error of our method, we could use that to generate the exact image. Instead, we verify our high-order algorithms on test problems for which the exact visualization is known in advance, and use that to verify that our algorithms produce the correct image.

We end by noting that it is not necessary to represent the high-order data directly to produce a pixel-exact image; with sufficient refinement, methods based on linear approximations to the model are also capable of producing these images. There are two main advantages to using the high-order data directly. First, pixel-exact images of object-space, linear methods are view-dependent, meaning that a linear

approximation sufficient to produce an accurate image from one view may not be sufficient from another. Second, visualizations produced with linear methods have to account for two different sources of error (visualization modeling error and algorithm error) before a pixel-exact image can be produced. Our methods, on the other hand, only have to consider the errors in the algorithm to produce pixel-exact images.

CHAPTER 4

ISOSURFACES

In this chapter we give an overview of our high-order isosurface rendering algorithm, which exploits the properties of spectral/hp element data to generate images in which the visualization error is both quantified and minimized [42]. This is accomplished by reducing the determination of the ray-isosurface intersection of spectral/hp element data to classic polynomial root-finding applied to a polynomial approximation obtained by projecting the finite element solution over element-partitioned segments along the ray. An overview of this procedure is given in Algorithm 1.

ALGORITHM 1: High-Order Isosurface Algorithm

Input: A ray $r(t)$, a list of isovalues ρ, and a list of all elements E traversed by the ray, ordered by intersection distance.

```
1  foreach Element E_i ∈ E do
2      Determine ray entrance t_a and exit t_b for element E_i.
3      Evaluate the field on the ray segment [t_a, t_b] using interval arithmetic to obtain
       field bounds [f_min, f_max].
4      if ∃_i : ρ_i ∈ [f_min, f_max] then
5          Project the field along the ray onto an n^th order polynomial p_n(t).
6          foreach ρ_i ∈ [f_min, f_max] do
7              Find all real roots r of p_n(t) − ρ_i = 0.
8              if ∃_i : r_i ∈ [t_a, t_b] then
9                  Report smallest r_j such that r_j ∈ [t_a, t_b].
10             end
11         end
12     end
13 end
```

4.1 Traversing the High-Order Mesh

The first step of our algorithm (Line 1) is to traverse the mesh for each ray. We start by casting the ray $\vec{r} : [0, \infty) \rightarrow \mathbb{R}^3, \vec{r}(t) = \vec{o} + t\vec{v}$ (where $\vec{o}$ denotes the ray origin and $\vec{v}$ denotes the ray vector) into the volume and find the closest element (i.e., the ray-element intersection has the smallest value of the ray parameter t). Finding this intersection, especially for a volume that consists of many elements, can take a long time. Therefore, once the first intersection is found, we use a modified version of the algorithm from Garrity [16] to move through the volume. Special care is required to do this correctly, as high-order elements can have curved faces, which can lead to a ray entering and exiting an element through a single face (Figure 4.1). Therefore, our modified algorithm considers the current face as well as the other element faces when searching for the next intersection point.

4.2 Range Estimation

Once we have identified the ray-segment that traverses an element, we next estimate the field's range along the ray (Line 3). This is an acceleration step based on the observation that not every ray segment will contain the isosurface. Since the root finding procedure described below is computationally expensive, we can achieve significant speedup by skipping elements which cannot contain the isosurface. We do this by using interval arithmetic (see Section 3.2) to give us an estimate of the function's range. If none of the requested isovalues falls within the range, then the segment can be safely skipped.

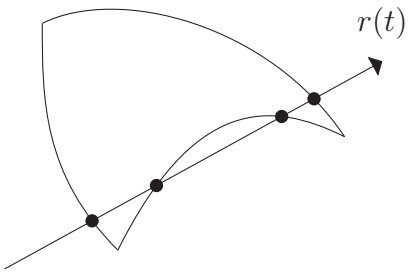


Figure 4.1. When elements are curved, rays can enter and exit an element through the same edge (pictured) or face.

4.3 Function Projection

Our goal of rendering an isovalue along a ray intersecting an element equates to finding the zeroes of a smooth function. Since polynomial root finding is far easier, in general, than finding the zeroes of a general function, we seek a means of representing the function along the ray within an element in world space as a polynomial with quantifiable and adaptability reducible error. This can be accomplished by finding the L_2 projection of the function onto a Legendre polynomial expansion (Line 5 of Algorithm 1), which takes the form:

$$u(x) = \sum_{n=0}^{\infty} \hat{u}_n P_n(x) \quad , \quad x \in [-1, 1]. \tag{4.1}$$

Here, $P_n(x)$ represent the n^{th} order Legendre polynomial.

The expansion coefficients $\hat{u}_n$ can be obtained as

$$\hat{u}_n = \frac{2n+1}{2} (u, P_n) . \tag{4.2}$$

To perform the function projection one can use Gauss quadrature rules such as the Gauss-Lobatto quadrature

$$\tilde{u}_n = \frac{2n+1}{2} \sum_{i=0}^{N} u(x_i) P_n(x_i) w_i, \tag{4.3}$$

where (x_i, w_i) represent the Legendre-Gauss-Lobatto quadrature nodes and weights, respectively (see [6]). This procedure is illustrated in Figure 4.2. The quadrature is exact if $u(x)$ is a polynomial of degree $2N - 1$ at most.

4.4 Root Finding

Given the projection to a polynomial described above, the ray root finding problem can now be formulated mathematically as the following problem: Find the smallest value $t \in [t_a, t_b]$ such that $p(t) - C = 0$ where C is the isovalue of interest (Step 7 of Algorithm 1).

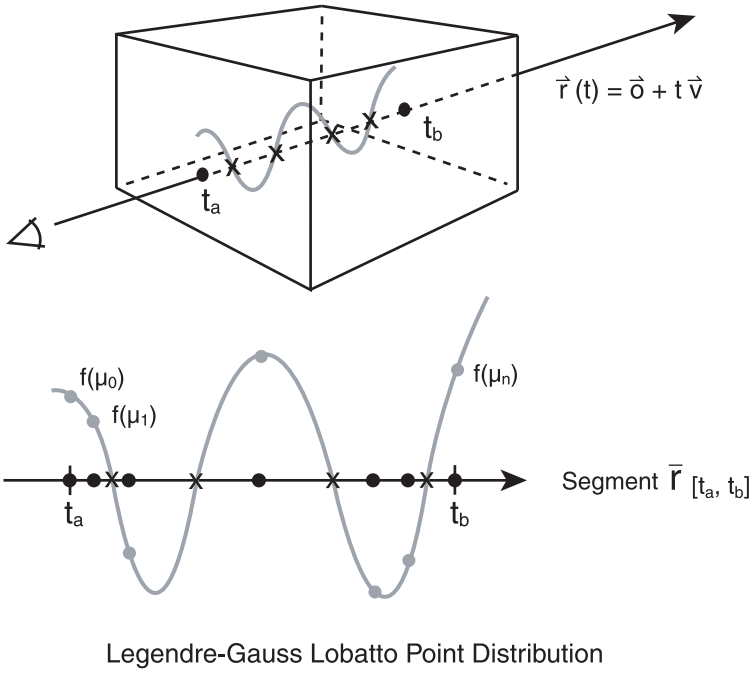


Figure 4.2. Schematic showing how a projected polynomial is formed along the ray. Once the entry and exit points within an element are known, the scalar field within the element $f(\vec{x})$ can be sampled along the ray using a Legendre-Gauss-Lobatto point distribution. A "best-fit" polynomial (in L_2) can be formed, upon which root finding can be accomplished.

Since we want to guarantee that we have found the smallest root in the interval $[t_a, t_b]$, we desire to find all possible roots, from which we can then easily ascertain the smallest one which lies within our interval of interest. Since we desire an algorithm which extends to arbitrary order, we utilize the mathematical idea of a *companion matrix* of a given polynomial. The linear algebra concept of a companion matrix provides us with the following result: given a polynomial of degree M, there exists a matrix $\mathbf{A}$ determined by the coefficients a_j (called the companion matrix of the polynomial $p(t)$) such that the eigenvalues of $\mathbf{A}$ provide the roots of the polynomial $p(t)$. Hence, instead of a root-finding problem *per se*, we must solve an eigenvalue problem. This concept naturally follows from the linear algebra concept of the characteristic polynomial of a matrix [21]; there is a rich history between root finding and eigenvalue solutions which we seek to exploit.

To help provide guidance as to how the companion matrix of a polynomial is

formed, we will demonstrate with respect to a sixth-order polynomial example. For example, consider the following polynomial:

$$p(t) = a_6 t^6 + a_5 t^5 + a_4 t^4 + a_3 t^3 + a_2 t^2 + a_1 t + a_0$$

where $a_j \in \mathbb{R}, j = 0 \ldots 6$, and where we assume that $a_6 \neq 0$.

The companion matrix for this polynomial is as follows [58]:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & -a_0/a_6 \\ 1 & 0 & 0 & 0 & 0 & -a_1/a_6 \\ 0 & 1 & 0 & 0 & 0 & -a_2/a_6 \\ 0 & 0 & 1 & 0 & 0 & -a_3/a_6 \\ 0 & 0 & 0 & 1 & 0 & -a_4/a_6 \\ 0 & 0 & 0 & 0 & 1 & -a_5/a_6 \end{pmatrix}.$$

This matrix is of upper Hessenberg form and hence a prime candidate for typical eigenvalue-finding techniques. Easily generated companion matrices exist for the monomial basis [58] and the Bernstein basis [65, 66]. The companion matrix associated with the Bernstein basis has been shown to be less prone to numerical precision errors. Given our polynomial $p(t)$ as a Legendre series, we can do a basis rotation to either the monomial or the Bernstein basis. Once this has been accomplished, a companion matrix can be formed, and the corresponding eigenvalue problem solved.

CHAPTER 5

CUT SURFACE VISUALIZATION

The behavior of a high-order scalar field along an arbitrary cut-surface can often be one of the primary questions that must be answered by a simulation. For example, a simulation of a proposed aircraft body may be performed to determine if it will survive the stresses of flying in a variety of scenarios. In such a simulation, users are interested in the behavior of the simulation at specific boundaries, such as the stress field along the aircraft's wing. An effective way to visualize these types of surfaces is through the application of color maps and/or contour lines (see Figure 5.1).

In this chapter, we describe a new set of methods for rendering color-maps and contour lines on arbitrary cut-surfaces (of which curved-element boundaries are of particular interest), extending the existing methods for applying color-maps to cut-

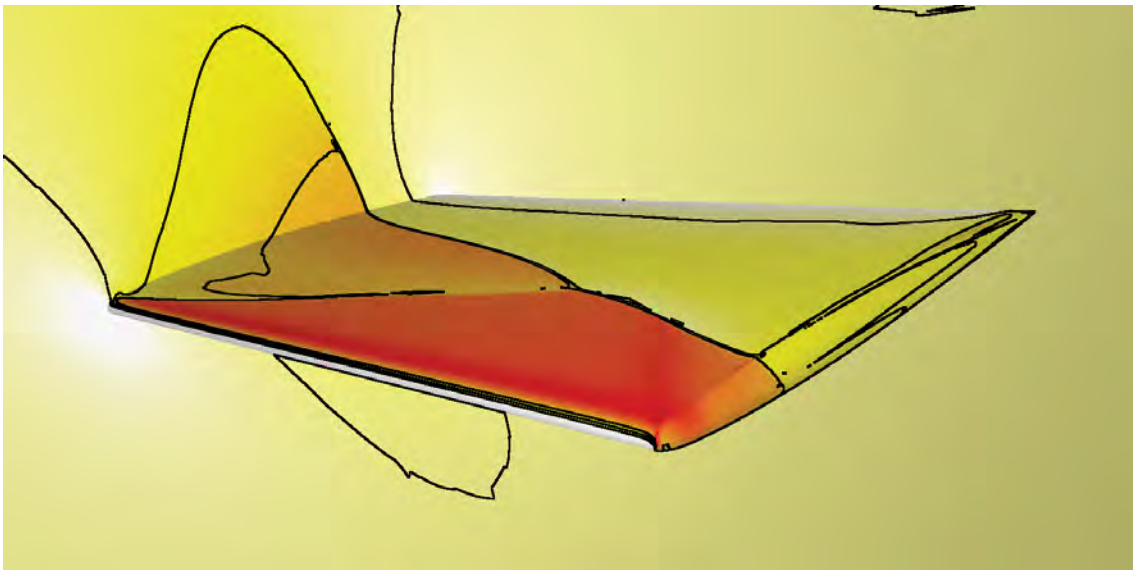


Figure 5.1. A surface rendering of the ONERA wing (see Section 7.4.3) rendered using the algorithm described in this chapter. The wing is represented by the curved element faces, while the plane is a cut-plane through the solution.

planes through high-order data [5, 17]. Our system's goal is to generate accurate and interactive images, allowing users to debug the simulation's code, accurately interpret the image and perform general exploratory visualization. The system uses the high-order data in its native state, without the need for low-order approximation, and uses the the knowledge of the structure and mathematical properties of the underlying fields to provide accurate images.

Our algorithm consists of two principle components: a surface sampling module and a color assignment module. The surface sampling module is responsible for casting rays to find the surface associated with each pixel, then sampling the field at the intersection point. The color assignment module is responsible for using the samples to generate a color map or contour line.

5.1 Surface Sampling Module

The surface sampling module is a ray generation program that first queries the downstream modules to determine the directions of the rays that are needed. It is assumed that a ray through the center of each pixel will be required, but any of the downstream modules can request additional rays. The module then casts rays through each of the requested locations.

The behavior of the ray once it finds an intersection depends on the surface type. This surface-dependent behavior is implemented using an OptiX closest-hit program. If the surface is a cut-surface then a secondary ray is cast to determine which element encloses the intersection point, and the scalar value at the point is then calculated. If the surface is an element face, then the scalar value can be calculated directly. Otherwise, the surface normal and color are determined. In this way, the scalar values of the field on the cut-surface are calculated and made available to the color mapping module, while at the same time, any surface geometry in the simulation can also be rendered.

5.1.1 Ray/Cut-Surface Intersection

Cut-surfaces are generally specified as either implicit 3D surfaces or two-parameter parametric surfaces. Implicit 3D surfaces are defined as the set of all points that satisfy the equation

$$f(x_1, x_2, x_3) = 0 \tag{5.1}$$

where $f : \mathbb{R}^3 \to \mathbb{R}$. The intersection between an implicit surface and a ray can be found by substituting Equation 3.14 into Equation 5.1, yielding:

$$\begin{aligned} g(t) &= f(r_1(t), r_2(t), r_3(t)) \\ &= f(O_1 + tD_1, O_2 + tD_2, O_3 + tD_3) = 0. \end{aligned} \tag{5.2}$$

Viewed in this way, the intersection test becomes a univariate root-finding problem. For simple implicit functions, this equation can be solved analytically. For example, a cut-plane is a simple implicit function of the form $f(x_1, x_2, x_3) = Ax_1 + Bx_2 + Cx_3 + D = 0$ which, after substitution, becomes a linear equation in t.

When the implicit surface is more complicated, such as when it is a high-order polynomial, it cannot be solved analytically. In these cases, numeric root-finding techniques are required [27]. In some cases, such as functions representing isosurfaces of high-order fields, the implicit function itself does not have an analytic form, and numeric techniques are required to both evaluate f and find the intersection [42].

The other type of surface of interest in a high-order setting is that of parametric surfaces, defined as $\boldsymbol{x} = p(u, v)$. If p is analytic, it may be possible to convert it into an implicit form, allowing for the direct use of methods already available for implicit forms.

A class of interesting parametric surfaces is that of the faces of the elements themselves. Faces that border simulation geometry are of particular interest because of their relationship to what happens at or near these locations (*e.g.*, pressure at specific locations on an aircraft's wing). Faces are defined as parametric functions:

$$\boldsymbol{x} = \boldsymbol{\Phi}(\xi_i, \xi_j) = \sum_a \sum_b \hat{u}_{ab} \phi_a(\xi_i) \phi_b(\xi_j), -1 \leq \xi_1, \xi_2 \leq 1 \tag{5.3}$$

with the ray-face intersection specified as the values for ξ_1, ξ_2, and t for which the following holds:

$$\boldsymbol{\Phi}(\xi_i, \xi_j) = \boldsymbol{r}(t). \tag{5.4}$$

The function $\mathbf{\Phi}$ is the mapping function described in Section 3.1 and does not, in general, have a conversion to a general implicit function.

5.1.2 Point Location

Once the cut-surface intersection point has been found, the next task is to determine the element in which it lies. We cast a new, secondary ray from the intersection point in a random direction. If the intersection point is inside the volume, then, no matter what direction we choose, the element corresponding to the closest ray-element intersection is the enclosing element. Since this is not true if the cut-surface intersection point is outside the volume, we constrain our cut-surfaces to lie entirely within the finite element volume.

5.1.3 Color Mapping Module

To apply a color map to a cut-surface, we sample the scalar field at the center of each pixel, using the procedures just described, and then use the resulting scalar value as a look-up into the color map.

5.1.4 Contouring Module

A pixel (i, j) belongs to the contour curve for isovalue ρ if it satisfies

$$\exists_{u,v} : P_{ij}(u, v) = \rho \tag{5.5}$$

where P is the scalar field of the cut-surface projected onto the image plane. As described in Section 3.1, the field is continuous over the pixel. Therefore, we can determine if the isovalue exists in the pixel by finding two points that bracket the isovalue:

$$\exists_{u_a,v_a,u_b,v_b} : P_{ij}(u_a, v_a) \leq \rho \leq P_{ij}(u_b, v_b). \tag{5.6}$$

Determining if the isovalue exists somewhere in the pixel can be a complicated and time consuming process that requires the determination of the global maximum and minimum scalar value over the pixel. To reduce the complexity of this test, instead

of checking the pixel's interior, our algorithm looks for the isovalue along a pixel's edge. This approach is attractive because it detects the same contours as searching the interior detects except for the case where the contour exists entirely in the pixel's interior (see the lower right corner of Figure 5.2). These types of contours will show up as isolated points in the image and will not help the user interpret the visualization, so the extra processing time to find them need not be taken.

A simple algorithm to determine if a pixel may be part of the contour is to perform point sampling at the four pixel corners. If the values at the corners bracket ρ, then the pixel can be marked as part of the contour with no further testing. This method is appealing for several reasons. First, it is fast. Sampling the pixel corners requires $(w+1)(h+1)$ samples, where w is the image width and h the height. This is only slightly more samples than are required for color mapping, which requires wh samples. Second, if the field is monotonic over the pixel, then this test is also accurate (i.e., it only marks pixels that are part of the contour and doesn't mark pixels that are not part of the contour). Sampling the endpoints of a monotonic function produces the function's range (see Figure 3.3(a)). Since the high-order field is not guaranteed to be monotonic over the pixel's edge, this algorithm cannot guarantee that it will find all contour pixels. It does, however, find a large percentage of them.

Because the field is not guaranteed to be monotonic, contours can take a variety of forms that will not be detected by the simple corner testing algorithm. As shown in Figure 5.2, contours can cross edges multiple times and span many pixels and still miss detection by the corner testing algorithm. We use interval arithmetic (see Section 3.2) to provide an estimate of the range.

For high-order fields, calculating the field's range over a pixel's side produces

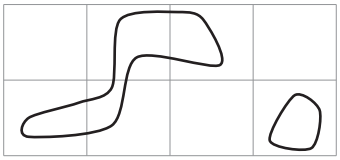


Figure 5.2. Contour lines that are not detected by simply checking if the pixel corners bracket the isovalue.

fairly good bounds, as can be seen in Figure 5.3, where we show that, even without performing any subdivisions, the number of pixels that may contain the isovalue is small. One factor that can contribute to wide interval extensions is the number of operations performed when evaluating an expression. An example of this can be seen in Figure 5.4, where the number of ambiguous pixels is compared between a 2^{nd}-order and 6^{th}-order data set. The 6^{th}-order data set has more ambiguous pixels than the 2^{nd}-order data set. Interval arithmetic provides conservative bounds that are obtained without using any derivatives, and are not negatively affected by the presence of extremal features.

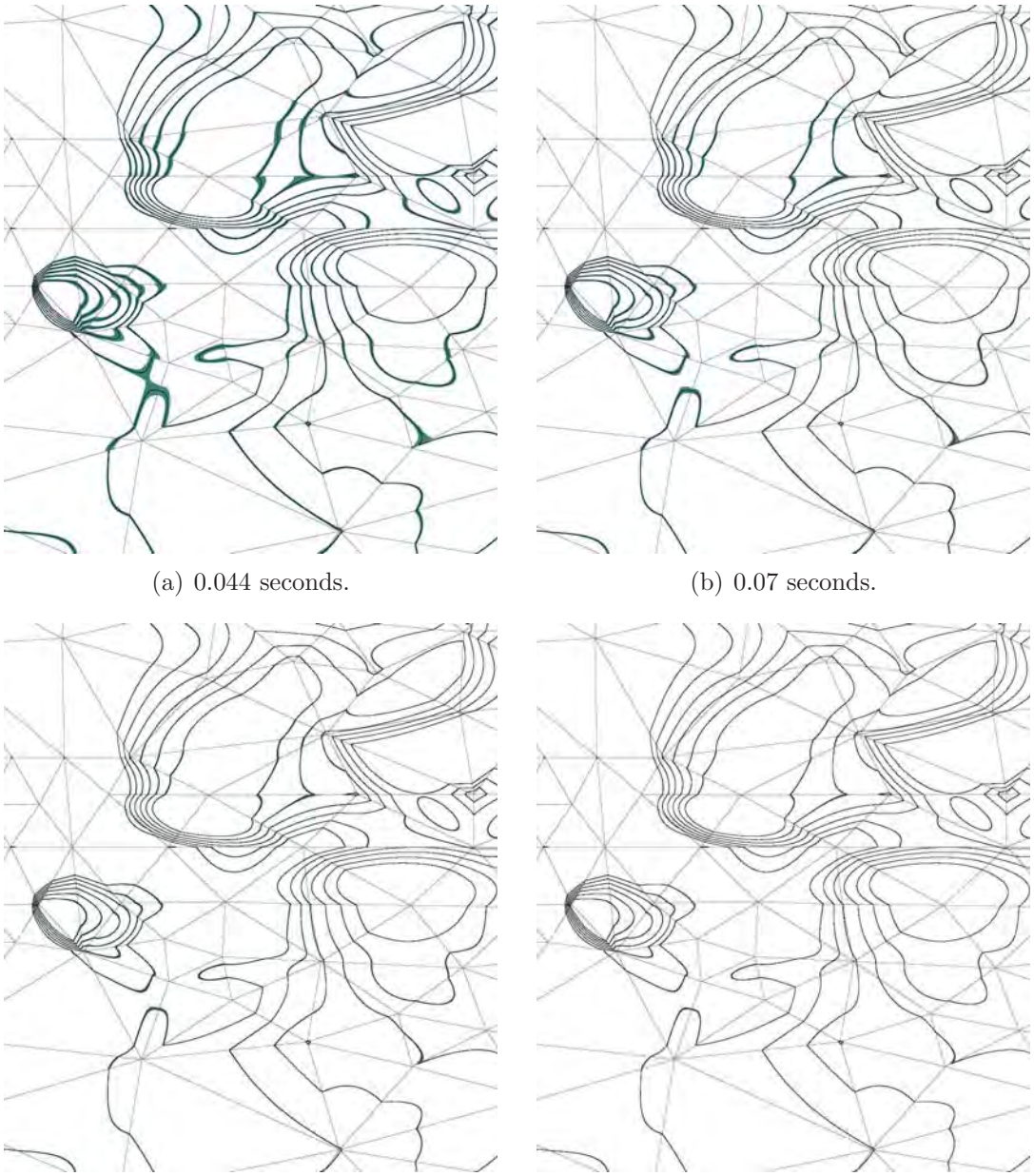
The cut-surface contouring algorithm proceeds as follows. First, sample the cut-surface at each pixel's corner. If any two pixel corners bracket an isovalue, mark the pixel. For each unmarked pixel, calculate the approximate bounds of the function along each side. If none of the contours fall within the approximate bound, reject the pixel. Otherwise, mark it as possible. Finally, for each possible pixel, subdivide the edge, taking additional samples as needed, to provide better bounds until a user defined tolerance is met. If contours fall outside the range, reject the pixel, otherwise accept.

Note that, when the algorithm completes, it is possible for some pixels to be marked ambiguously. This cannot be avoided unless the global minimum and maximum over the pixel is calculated. Our system allows for these ambiguous pixels to be colored a different color. The user can then increase the amount of subdivision as needed to reduce the number of ambiguous pixels. In practice, we have found that dividing a pixel's side into eight subintervals is enough to handle most ambiguities when dealing with data sets up to sixth order.

5.2 Results

The effectiveness of the methods described in this chapter are illustrated by applying them to two high-order fluid flow data sets. Note that we will frequently refer to n^{th}-order volumes or elements as a shorthand for saying that the field is represented by n^{th}-order polynomials, in tensor space, in each direction.

The first example consists of incompressible flow past a block with an array of



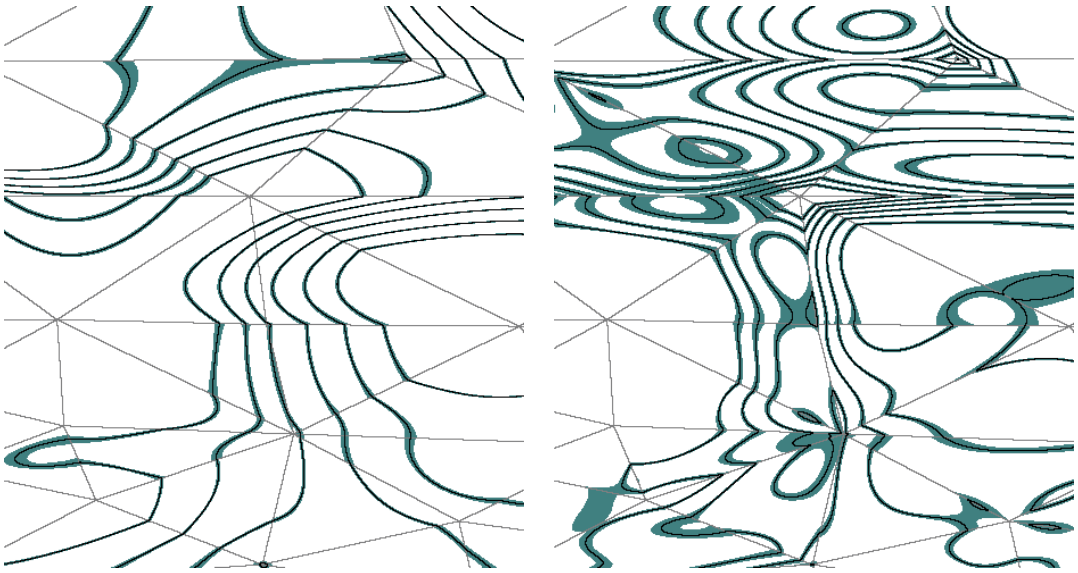
(a) 0.044 seconds.

(b) 0.07 seconds.

(c) 0.11 seconds.

(d) 0.18 seconds.

Figure 5.3. Contours generated on a cut-plane of the block/plates data set (see Section 5.2). Pixels that contain the isovalue are marked in black. Pixels that cannot contain the isovalue are white, and pixels that may contain the isovalue are teal. In 5.3(a), only corner testing has been performed. In 5.3(b), one level of subdivision has been performed. Two levels were performed in 5.3(c), and three in 5.3(d). We can see that additional testing reduces the number of ambiguous pixels. Rendering times for a 1000×1000 image are given underneath each image. While subdivisions take extra time, the rendering times are still interactive.



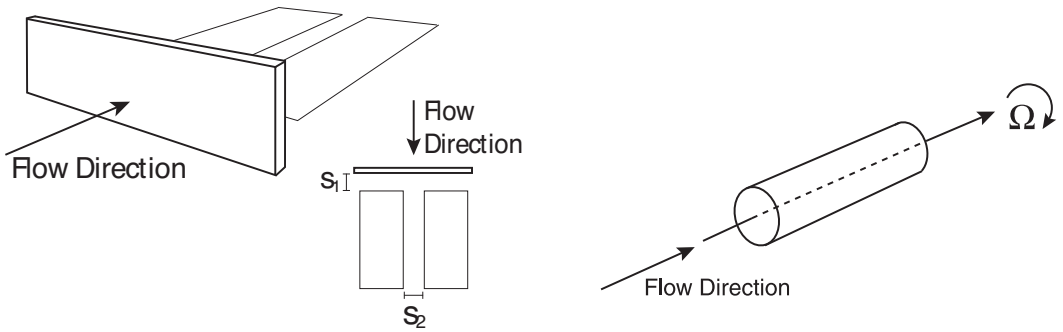
(a) 2^{nd}-Order (b) 6^{th}-Order

Figure 5.4. Comparison of ambiguous pixels (teal) between a 2^{nd}-order (left) and 6^{th}-order (right) data set.

splitter plates placed downstream of the block. A schematic of this regime is presented in Figure 5.5(a). As the fluid impinges upon the block, it is diverted around the structure, generating vorticity along the surface. For the purposes of this paper, we will focus our attention on a configuration consisting of a plate spacing of one unit (nondimensionalized with respect to the block height). The 3D computational mesh consists of $3,360$ hexahedra and $7,644$ prisms. All simulations were performed at a Reynolds Number (Re) of 200.

The second data set consists of a rotating canister traveling through an incompressible fluid. A schematic of the flow regime under consideration is presented in Figure 5.5(b). The 3D mesh consists of $5,040$ hexahedra and 696 prisms, with the computational problem being solved using third-order polynomials within each element. The solutions presented herein were computed at $Re = 1000$ and with an angular velocity of $\Omega = 0.2$.

All tests described in this section were performed on a desktop workstation equipped with an NVIDIA Tesla C2050 GPU and Intel Xeon W3520 quad-core processor running at 2.6 GHz. All code run on the GPU was implemented in OptiX. Code executed on the CPU was written as single-threaded C++ code. The GPU algorithms were run using 32 bit floating point precision, while the CPU code was run using 64 bit floating point precision. In our tests, there is a negligible performance impact between 32 and 64 bit precision on the CPU, while 64 bit precision on the GPU generally



(a) Block/Plate Configuration (b) Canister Configuration

Figure 5.5. Schematic showing the basic block/splitter plate (left) and rotating canister (right) configurations under consideration.

doubles overall execution time. As we will show when we discuss performance, our methods execute at 10-20 frames per second even for large image sizes, so 64 bit precision can be used when necessary without significantly impacting performance.

5.2.1 Linear Comparison Models

Visualizations of high-order data are traditionally performed using linear primitives. Because of this, we will contrast the performance, accuracy and resource consumption of our methods to the commonly used linear algorithms. We will show that while linear methods can produce acceptable images under the right circumstances, a more effective way to reliably achieve accurate visualizations (under reasonable resource constraints and without intervention) is through the methods described here. The linear test cases were implemented using the Visualization Toolkit (VTK) [52] because it is a well-known system, in current use, and therefore provides a good foundation on which to base our tests.

Two approaches were used for representing the high-order data with the linear structures found in VTK. In the first approach we sampled the entire high-order volume onto a 3D regular grid of points, where the spacing between points is constant. Values between grid points are calculated using linear interpolation. For the second approach a triangular mesh is created and sampled with the high-order data at the triangle vertices. As with the first volume, data values between vertices are obtained via linear interpolation. Sampling was performed using a CPU-based implementation. We used an octree data structure to accelerate the location of the element enclosing a given sample point. Timings for sampling several high-order volumes onto a grid are found in Figure 5.6(a), and the timings to sample a cut-plane running through the center of the data set is in Figure 5.6(b).

While sampling onto a 3D grid is the most flexible approach (since so many visualization algorithms can be applied), it is also the most time consuming and quickly becomes prohibitive, especially as the volume order increases. Sampling directly onto the cut-plane, while not as flexible, is fast enough to be practical. The problem with sampling onto a triangular mesh is that the memory required to store the mesh and samples grow roughly in a squared manner (by the spacing). We can

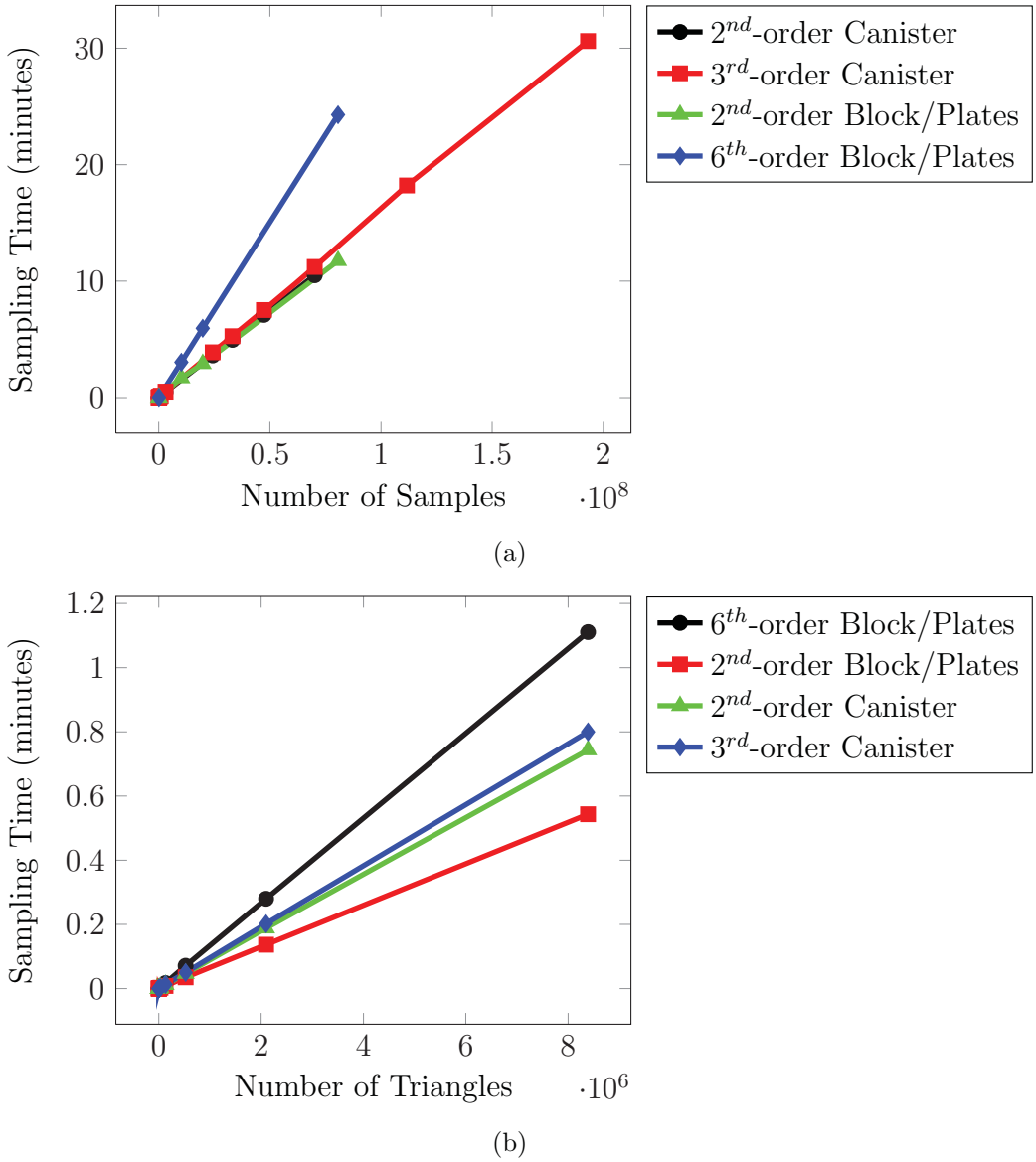


Figure 5.6. Timings to sample high-order volumes onto linear data structures. (a) - Time to sample a grid of evenly spaced points that span the high-order volume. (b) - Time to sample a triangular mesh representing a cut-plane through the middle of the volume.

produce a cut-plane with $8,388,608$ samples using the 2^{nd}-order block/plate data set in about 30 seconds, but it consumes 180 MB of memory, compared to 4 MB to represent the entire high-order volume. And, as shown in the examples below, there are not enough samples to produce highly-accurate visualizations.

In the remainder of this section, our comparison tests are performed with the triangular mesh-based technique instead of the 3D grid-based approximation. We found that the quality of the grid-based approach is worse (sometimes significantly) than the mesh approach because the cut-surface may span between the grid samples, meaning that the contours and colors produced are interpolated from samples that do not actually lie on the surface. Using the triangular mesh, however, guarantees that all samples are on the cut-surface, providing more accurate images and better comparisons.

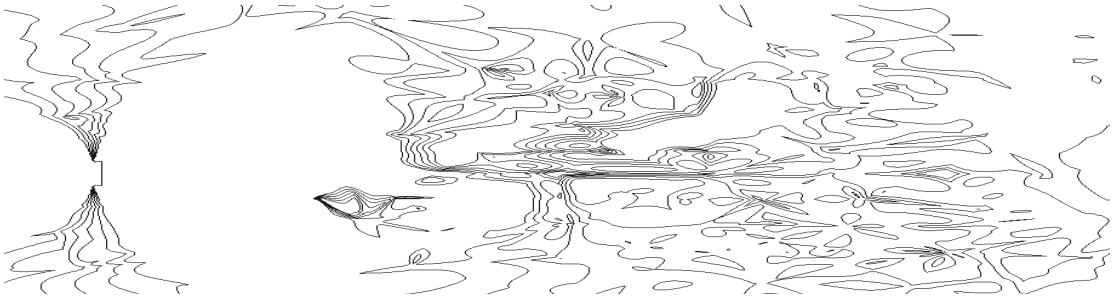
5.2.2 Contouring

While there are contouring algorithms available for high-order methods (as discussed in Section 2.3), we are unable to provide direct comparisons with those methods because they either are restricted by the type of element or the maximum order. Contours for the linear data were produced by the vtkContourFilter. The results are shown in Figure 5.7.

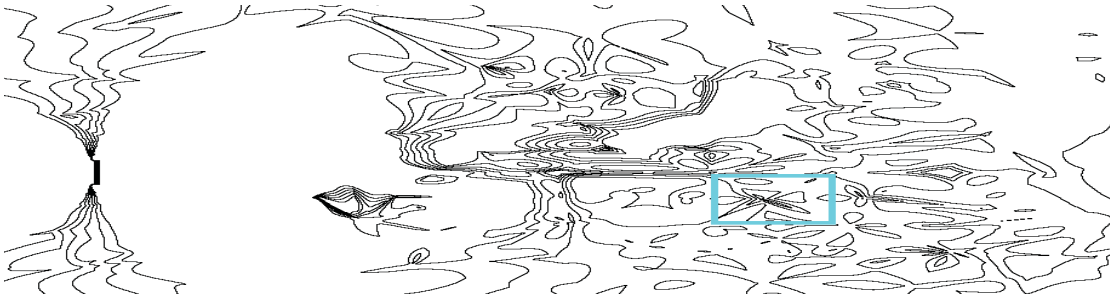
The linear approximation in Figure 5.7(b) is a significant improvement when compared to the more coarsely sampled volume, and it is difficult to see much difference between this image and our high-order method. However, this is only true at the current resolution, and if we zoom in to the image (Figures 5.7(d) and 5.7(e)), we can see that there are still significant errors present. While further sampling can improve the generated contour, there are limits to the sampling resolution (due to the amount of resources consumed). And importantly, the method described in this paper performs at about the same speed as the coarsely sampled image.

In the coarsely sampled contour image (Figure 5.7(a)), there are many errors that have been noted with red circles: incorrect topology, missed contours and incorrect shapes. What is interesting is that it is not possible to determine if these contours are accurate from the image alone. We need either a cut-plane with greater resolution or

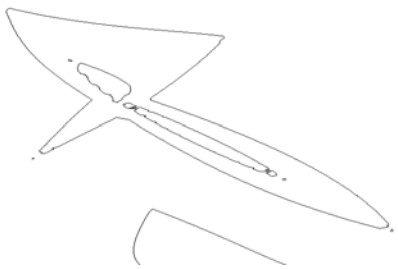
(a) 524, 288 Triangles. VTK Contour Generation Time = 0.265 seconds.



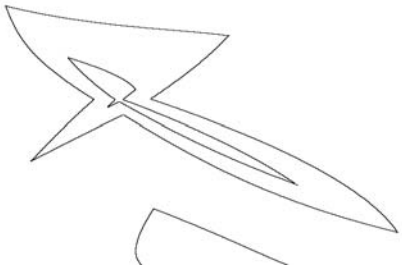
(b) 8, 388, 608 Triangles. VTK Contour Generation Time = 3.5 seconds.



(c) High-Order Rendering. Rendering time for 2000×2000 image = 0.3 seconds.



(d) Detail View of 5.7(b)



(e) Detail View of 5.7(c)

Figure 5.7. Comparison between pressure contours of the block/plane data set generated using linear methods (5.7(a) and 5.7(b)) and our high-order method (5.7(c)). The approximation in 5.7(a) has several significant errors, marked in red, which disappear when using a smaller sampling. The highlighted area in 5.7(c) is shown in more detail in 5.7(d) for linear interpolation and 5.7(e) for our system.

an image from our system to notice the inaccuracy.

5.2.3 Color Maps

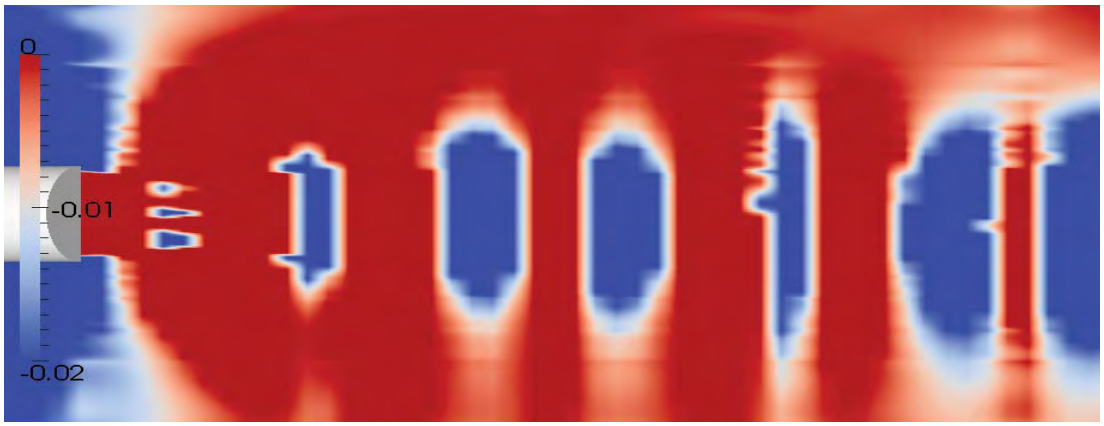
We next show show a comparison between linear approximations of a cut-plane through the canister data set with our algorithm. Since the cut-plane is an inherently linear cut-surface, there is no surface approximation error, so differences between our method and the linear methods are due solely to the differences between linear and high-order interpolation (see Figure 5.8).

As expected, the images get progressively closer to the image generated by our algorithm as the number of samples increase. However, even at the very fine sampling (shown here), there are still subtle errors. Additionally, the memory required to store the mesh is large which requires a lengthy prepreprocessing step. It is interesting to note that, unlike the contour images, it can be visually obvious when a color map needs to be further refined by the presence of sharp and "boxy" gradients.

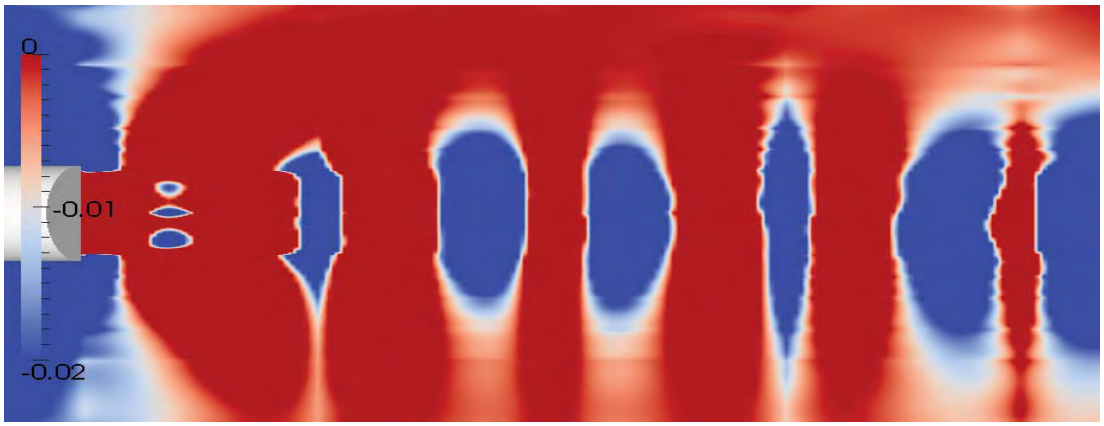
5.2.4 Performance

For the methods described in this paper to be useful, they must not only be accurate but interactive as well. For each test, we rendered a view of a cut-plane and cut-cylinder where the entire image was covered. Since the rendering pipeline is restarted from scratch with every view change, there was no need to run the timing tests with a variety of viewing changes. We executed the tests 100 times and reported the average time to render the entire scene.

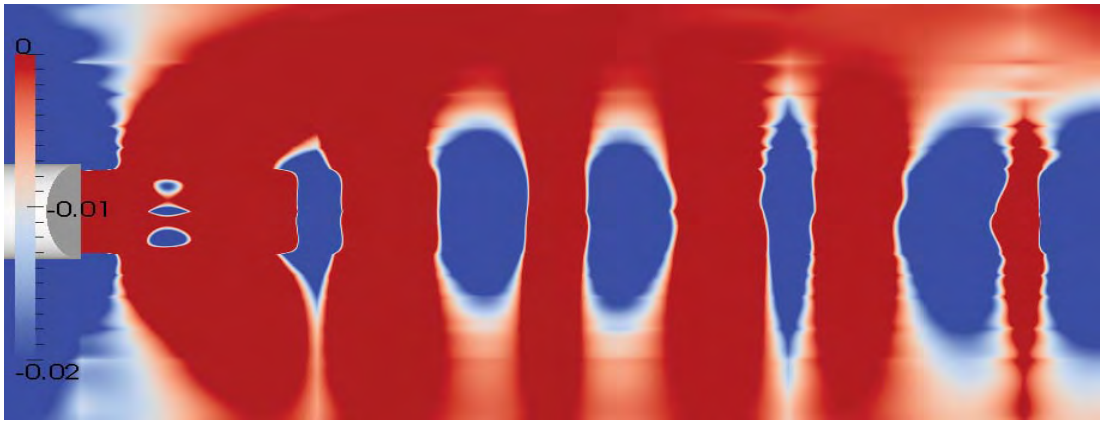
Since we did not have fluid flow simulation data higher than 6^{th}-order, synthetic data sets were generated to obtain the timings. We were interested in how timing was related to the simulation's order, number of elements, and the overall image size. In Figure 5.9, we can see that the most important factor governing the performance of our system is the image size. This figure also illustrates the impact of using more complicated cut-surfaces, specifically bicubic patches. While overall rendering time is increased with more complicated surfaces, it is still interactive. This has several positive implications. First, if the system is not as interactive as desired, additional speed can be gained by creating smaller images. Second, the system is capable of handling large high-order data sets interactively, and there is no indication of an



(a) Cut-plane with 902,289 triangles, VTK Rendering Time = 0.08 seconds.



(b) Cut-plane with 8,388,608 triangles, VTK Rendering Time is 2.0 seconds.



(c) Pixel-exact cut-plane color map. Rendering time is 0.015 seconds for a 1800×800 image.

Figure 5.8. Color maps for a cut-plane through the canister data set with a coarse sampling (5.8(a)), a fine sampling (5.8(b)), and pixel-exact using our method (5.8(c)).

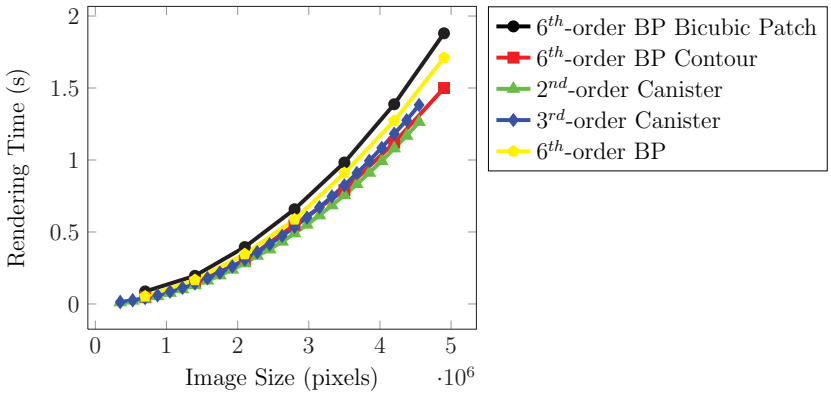
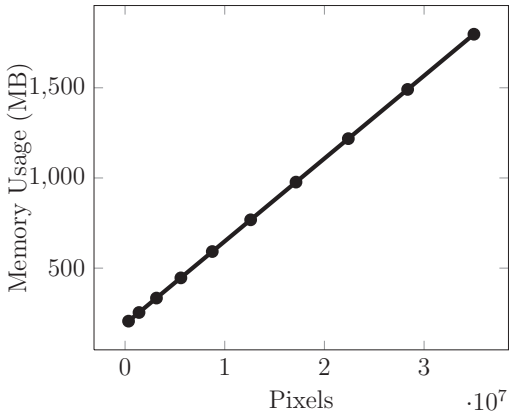


Figure 5.9. Time in seconds to render high-order data sets for a variety of image sizes and cut-surfaces, measured in pixels. BP refers to the block/plate configuration.

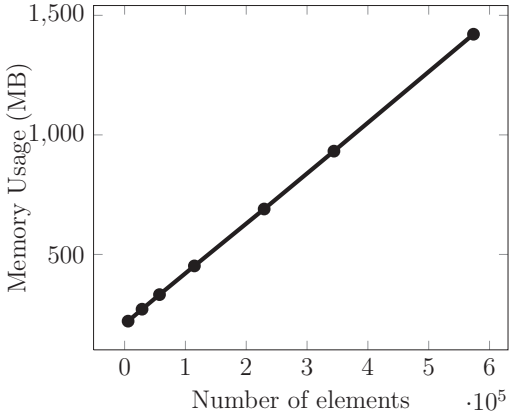
upper limit to the number of elements that can be supported (except for the size of the native solution that must fit in the GPUs memory).

Additional timing results are shown in Figure 5.10 and, unless specifically noted otherwise, are valid for both the contouring and color mapping module (as they generally take the same amount of time). In Figure 5.10(c), it can be seen how increasing the field's order impacts performance. If the order is increased significantly, then it will impact performance negatively, but for typically-used orders, around 2-14, rendering times are not significantly impacted. In Figure 5.10(d), we tested our system on large data sets with up to 200,000 elements and it can be seen that our system's speed is not strongly related to the number of elements.

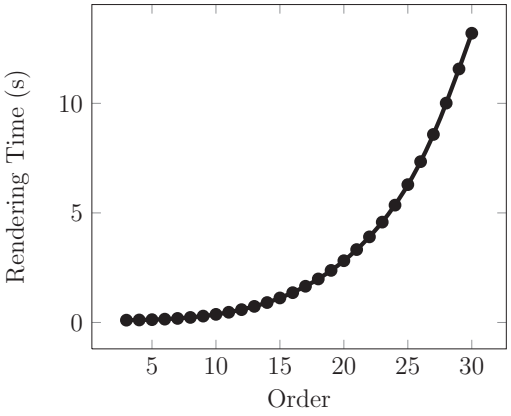
In Figures 5.10(a), 5.10(b) and 5.11 we show the GPU memory required by our system. We start with a $5,000$ element, 8^{th}-order volume rendered in a 1000×1000 image, then vary the number of elements, polynomial order, and image size. These graphs show that our system is capable of storing large volumes of high order on a single GPU. Increasing the image size results in a linear growth in memory usage due to the constant amount of per-pixel memory required for calculation and rendering. Increasing the number of elements also scales linearly, as the number of coefficients per



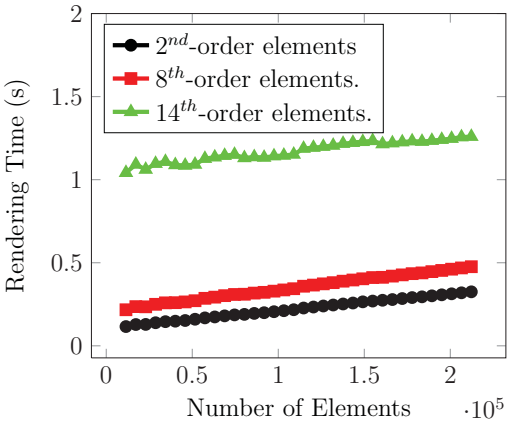
(a) Varying image size with $50,000$ 8^{th} order elements.



(b) Varying the number of elements with 8^{th} order solutions and 1000×1000 image size.



(c) Time in seconds to render volumes with increasing polynomial order.



(d) Time in seconds to render volumes with a varying number of elements.

Figure 5.10. Performance results and memory usage for rendering high-order data using our system. For typical usage scenarios, the parameter that impacts performance the most is image size.

element is constant for a given polynomial order. When increasing the polynomial order, the number of coefficients required to support the solution grows as $O(n^3)$, where n is the polynomial order, leading to the memory growth shown in Figure 5.11.

5.3 Summary

The algorithms presented in this chapter were motivated by the lack of existing visualization techniques capable of interactively and accurately rendering color-maps and contour lines on arbitrary cut-surfaces. We have described new algorithms that are capable of rendering surfaces interactively and accurately while using the high-order data in its native form (i.e., we do not need to resample onto lower-order constructs). We have also shown that the most important factor for determining rendering speed is the size of the final image, indicating that our system can efficiently handle high-order data sets with a large number of elements with a large number of modes. An additional benefit is that these interactive frame rates are achievable on commodity GPUs, meaning simulation scientists can easily perform even the most demanding visualizations at their workstations.

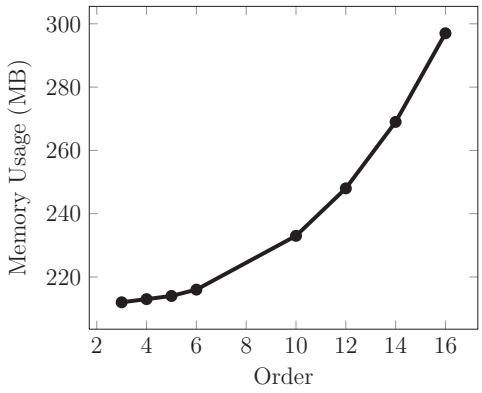


Figure 5.11. Varying order with $50,000$ elements and a 1000×1000 image.

CHAPTER 6

VOLUME RENDERING

The direct volume rendering of the high-order fields produced by spectral/*hp* finite element methods using linear primitives (e.g., those used by existing volume rendering packages such as Voreen [37], ImageVis3D [1] and VTK [52]) is not possible since these primitives are unable to directly represent the high-order field. In order to use these systems, engineers must first create compatible linear approximations of the high-order data. Since linear approximations do not, in general, faithfully represent high-order data, this approximation step introduces error into the visualization pipeline. This error can be reduced by creating linear approximations with smaller spacing between samples, but this comes at the expense of increased computation time to sample the high-order data and increased memory usage to store the resulting data set. In practice, it is difficult if not impossible to eliminate the error introduced by linear approximations through the use of increased sampling.

In this chapter, we develop a new direct volume rendering method that uses the properties of spectral/*hp* finite element fields to produce images that are both *accurate* and *interactive*. We obtain accuracy by using the properties of spectral/*hp* elements to categorize each ray based upon the properties of the transfer function composed with the scalar field. We prove that, in the optimal case of fields that are smooth along the ray, the evaluation of the volume rendering integral will generally exhibit second-order convergence, with a worst-case of first-order convergence. We use these properties to develop an optimized high-order volume rendering algorithm, in which we first categorize each ray based on the local properties of the transfer function and scalar field, then evaluate the volume-rendering integral with a quadrature method optimized for the category. While our primary motivation is the desire to generate pixel-exact images, which we define as an image which does not change with addi-

tional refinement of the volume rendering integral, performance is also an important consideration for a usable system. We have therefore implemented our system on the GPU, using a combination of NVIDIA's OptiX [45] ray-tracing framework and Cuda.

6.1 Background and Overview

We use the *emission-absorption* optical model [34] for direct volume rendering, in which the irradiance along a ray segment is given by:

$$I(a,b) = \int_a^b \kappa(f(t))\ \tau(f(t))\ e^{-\int_a^t \tau(f(u))\ du}\ dt \tag{6.1}$$

where a and b are the segment endpoints, κ and τ are the user-defined emissive (color) and density transfer functions, respectively, and $f(t)$ is the scalar field at a point t along the segment. We refer to the integral $-\int_a^t \tau(f(u))\ du$ as the *inner* integral and I as the *outer* integral.

This integral cannot be solve analytically except in the case of linear tetrahedra [64]. It is common, therefore, to use numerical integration techniques to evaluate it. Evaluating the integral is often the bottleneck in the volume rendering process. A commonly used strategy for accelerating the integral's evaluation is through the use of preintegration [12, 11]. The goal of preintegration is to perform as much of the work of evaluating the integral as possible in a preprocessing step, and turn the task of evaluating the integral into one of a table lookup. Given two samples s_0 and s_1 along a ray, and an assumption that the field varies linearly between these samples, the result of the volume rendering equation can be computed in a preprocesing step and stored in a 2D table, indexed by the two sample points. This table is often stored as a texture on the GPU, allowing efficient lookup of the result. This approach does not extend to high-order fields, however, since the assumption of a linear field between sample points does not hold. To obtain accurate preintegrated tables for high-order data, each segment along the ray must be represented by n samples points, where n is the order of the field. This results in an n dimensional lookup table, which becomes intractable for anything over a 2^{nd}-order field. We must therefore find other ways of accelerating the integral's evaluation.

While there are no theoretical constraints on a transfer function's form, we assume that transfer functions are piecewise-linear functions. While transfer functions using different bases exist (e.g., Gaussian basis functions [26]), piecewise-linear transfer functions are consistent with the transfer function interfaces provided by many existing volume rendering systems [18, 37, 1]. We limit our attention to the case of one-dimensional transfer functions.

Piecewise-linear transfer functions are specified in terms of a finite number of points $(c_0, T_0), (c_1, T_1), \ldots (c_n, T_n)$, where c_i is called a *breakpoint* and T_i is the corresponding value of the transfer function at that point. Breakpoints correspond to the transitions between linear pieces in the transfer function. While the transfer function is continuous at the breakpoints, the derivative is undefined at these points.

Because the transfer function has breakpoints where the derivative is undefined, so does the composition of the transfer function with the scalar field along the ray, $\kappa(f(t))$ and $\tau(f(t))$. Given a smooth function f, the result of composing the transfer function with f is a piecewise-smooth function which is only C^0 at a finite number of breakpoints. This is applicable to both CG and DG methods because, as will be discussed in Section 6.2, we evaluate the volume rendering integral on a per-element basis.

Effective evaluation of the volume rendering integral requires proper accounting of the breakpoints. Ideally, if the breakpoints can be found [64], we could then evaluate the volume rendering integral as follows (with t_i being the i_{th} breakpoint):

$$I = \sum_{i=0}^{n} \int_{t_i}^{t_{i+1}} \kappa(f(t))\tau(f(t)) e^{-\sum_{j=0}^{i-1} \int_{t_j}^{t_{j+1}} \tau(f(u))\, du - \int_{t_i}^{t} \tau(f(u))\, du}\, dt \tag{6.2}$$

where the integral of each segment, being smooth, could then be evaluated using high-order methods such as Gaussian quadrature [8]. While this approach is conceptually appealing, the high-order nature of the field makes it inefficient in practice. Finding n breakpoints along the ray is equivalent to finding the isosurfaces associated with each breakpoint. While this can be done, it is too computationally expensive for our interactivity requirements [42].

Since we do not know the location of each breakpoint, we cannot assume that the use of high-order quadrature methods will automatically lead to high-order convergence. The convergence properties of high-order quadrature methods assume smooth functions, which is an assumption that is violated by the existence of breakpoints.

6.2 High-Order Volume Rendering

We begin our discussion by providing an overview of our high-order volume rendering algorithm (Algorithm 2), followed by our analysis of the volume rendering integral for piecewise-smooth functions that has motivated it.

Steps 1 and 2 represent traversal of the finite element grid and is performed in the same way as the traversal discussed for isosurfaces in Chapter 4. The contribution of

ALGORITHM 2: High-Order Volume Rendering Algorithm

Input: A ray $R(t)$, color transfer function κ, density transfer function τ, a list of all transfer function breakpoints s, and a list of all elements E traversed by the ray, ordered by intersection distance.

1 **foreach** *Element* $E_i \in E$ **do**
2 Determine ray entrance t_a and exit t_b for element E_i.
3 Evaluate the field on the ray segment $[t_a, t_b]$ using interval arithmetic to obtain field bounds $[f_{min}, f_{max}]$ and transfer function bounds $[\kappa_{min}, \kappa_{max}]$, $[\tau_{min}, \tau_{max}]$.
4 Classify each transfer function into one of three categories based on these bounds: empty (**E**), piecewise-smooth (**PS**), or smooth (**S**).
5 Integrate each segment based on the classification (details are given in Section 6.4).
6 **if** $\tau \in \mathbf{E}$ *(empty space)* **then**
7 Skip this ray segment.
8 **else if** $\kappa \in \mathbf{E} \wedge \tau \in \mathbf{S}$ *(nonemitting occlusion)* **then**
9 Integrate τ using Gauss-Kronrod quadrature (Section 6.3.3), with the number of sample points chosen based on the order of E_i.
10 **else if** $\kappa \in \mathbf{E} \wedge \tau \in \mathbf{PS}$ **then**
11 Integrate τ using trapezoidal rule (see Section 6.3.1).
12 **else if** $\kappa \in \mathbf{PS} \wedge (\tau \in \mathbf{PS} \vee \tau \in \mathbf{S})$ **then**
13 Evaluate inner and outer integrals using trapezoidal rule (see Section 6.4.3).
14 **else**
15 Evaluate inner integral using trapezoidal rule and outer integral using Gaussian quadrature (see Section 6.4.3).
16 **end**
17 **end**

this work begins on Step 4. In this step, we classify the transfer functions over $[t_a, t_b]$ into one of three categories based on the structure of the field over the segment: empty space segments (**E**), where the transfer function is zero along the entire segment; piecewise-smooth segments (**PS**), where the segment contains one or more breakpoints; and smooth segments (**S**), where the segments contains no breakpoints. We discuss how we use these categories to choose an appropriate quadrature method for the volume rendering integral in Section 6.4.

Categorization of a ray segment requires the range of the scalar field along the ray, from which we can determine if the transfer function is zero along the entire segment or if it contains any breakpoints. One approach we can use is to calculate the global min and max using standard optimization techniques. While this approach will work, and will produce accurate segment categorization, it is a computationally expensive operation that does not facilitate interactivity. Instead, we generate fast and conservative estimates of the range through the use of interval arithmetic [39] (see Section 3.2). The implication of this choice is that we may perform more work than needed during integration, but save the time computing the exact range of the function. As we discuss in further detail below, this is a reasonable trade off for current GPU architectures.

We now discuss how we use interval extensions to categorize rays. Let $f(t)$ be the field along a ray segment on the interval $X = [t_a, t_b]$. We first construct the interval extension $F(X)$ (giving us an estimate of the true range of f) using interval arithmetic as described above. Let $c_i, i \leq n$ be the n breakpoints associated with the density transfer function τ. We categorize τ as follows:

$$C(\tau) = \begin{cases} E & \text{if } F(X) = [0,0] \\ PS & \text{if } \exists_i : c_i \in F(X) \\ S & \text{if } \forall_i : c_i \notin F(X). \end{cases} \tag{6.3}$$

The categorization for each of the color channels proceeds in an analogous fashion. It is important to note that this categorization is not precise, meaning that a ray can be categorized as a piecewise-smooth segment when it is actually smooth. This is because the true range is a subset of the estimated range, so a breakpoint can be

in the estimated range while falling outside the actual range. The implication of this is that we may handle some rays suboptimally by using the rules developed below for piecewise-smooth functions, rather than the more optimal approach for smooth functions. In this context, suboptimal means that, by classifying a smooth function as piecewise-smooth, we will use a lower-order quadrature scheme and therefore will require more samples to achieve a given level of accuracy. If tighter bounds on the range estimate are desired, the ray segment can be broken into subsegments to produce a tighter bound [41].

6.3 Integration Techniques

In this section, we provide the motivation for the integration methods we use to evaluate the volume rendering integral. As we mentioned in Section 6.1, the field along the ray is a smooth function, and seems to be a perfect candidate for the use of high-order quadrature rules. However, as we show below, the existence of breakpoints along the ray, which are introduced through the composition of the field with the transfer function, limits us to linear or quadratic convergence, even when using high-order quadrature rules such as Gaussian quadrature.

As indicated in Section 6.1, one way to address the presence of breakpoints along the ray is to use a root-finding procedure to find the location of each breakpoint, then to use high-order quadrature on the smooth segments between breakpoints. While this approach is appealing in theory, it is not useful in practice due to both the number of breakpoints found in typical transfer functions (increasing the number of root-finding calls needed) and the high-order field along the ray (increasing the time and complexity of the root-finding routine). Therefore, in the sections that follow, we focus our attention on the case where the location of each breakpoint is unknown, and only coincides with the location of a sample point through chance.

6.3.1 Quadrature of Piecewise-Smooth Segments

When the transfer function along a ray segment is piecewise-smooth (i.e., the segment contains at least one breakpoint), we cannot use high-order quadrature routines and expect high-order convergence. This is because the convergence analysis for these methods assumes continuity in the function's derivatives. We now discuss the

convergence behavior of an arbitrary n-point quadrature rule when used to estimate the integral of a piecewise-smooth function.

Let $f(t)$ be piecewise-smooth in $[a, b]$ with a single breakpoint at $c \in (a, b)$:

$$f(t) = \begin{cases} e(t) & t \leq c \\ g(t) & t > c \end{cases} \tag{6.4}$$

where $e(c) = g(c)$, $e'(c) \neq g'(c)$, $e \in C^{\infty}[a, b]$ and $g \in C^{\infty}[a, b]$. We consider n-point quadrature methods of the form

$$\int_a^b f(t)\, dt \approx \sum_{i=1}^{n} w_i f(t_i) \tag{6.5}$$

where $a \leq t_i \leq b$, and $\sum_{i=1}^{n} w_i = b - a$. The abscissas can be evenly-spaced (corresponding to Newton-Cotes quadrature) or nonequally spaced (such as the case of Gaussian quadrature). The error between the integral's true value and this approximation is given by

$$E = \int_a^b f(t)\, dt - \sum_{i=1}^{n} w_i f(t_i). \tag{6.6}$$

To determine if high-order quadrature is effective for piecewise-smooth functions, we wish to determine how this error behaves as the interval $h = b - a$ becomes smaller. We will do this by rewriting each term in Equation 6.6 using Taylor's theorem, expanding e and g around the breakpoint c, which gives

$$f(t) = \begin{cases} e(t) = T_e(t) = e(c) + e'(c)(t-c) + e''(\xi_e)\frac{(t-c)^2}{2} & t \leq c \\ g(t) = T_g(t) = g(c) + g'(c)(t-c) + g''(\xi_g)\frac{(t-c)^2}{2} & t > c, \end{cases} \tag{6.7}$$

for some $\xi_e \in (t, c)$ and $\xi_g \in (c, t)$. Using these series to rewrite the integral gives

$$\begin{aligned} &\int_a^b f(t)\, dt = \int_a^c T_e(t)\, dt + \int_c^b T_g(t)\, dt = \\ &(c-a)e(c) - \frac{(a-c)^2}{2}e'(c) - \frac{(a-c)^3}{6}e''(\xi_e) + \\ &(b-c)g(c) + \frac{(b-c)^2}{2}g'(c) + \frac{(b-c)^3}{6}g''(\xi_g). \end{aligned} \tag{6.8}$$

Since $e(c) = g(c)$, this then becomes

$$\int_a^b f(t)\,dt = he(c) - \frac{(a-c)^2}{2}e'(c) + \frac{(b-c)^2}{2}g'(c) + O(h^3). \tag{6.9}$$

To approximate the integral, assume we use an n-point quadrature formula spanning the entire domain of integration where $n \geq 2$ and where we are guaranteed that the breakpoint c lies between two sample points, $c \in (t_k, t_k + 1)$. The quadrature approximation of the integral can then be written as:

$$\begin{aligned} Q_n &= \sum_{i=1}^{k} w_i T_e(t_i) + \sum_{i=k+1}^{n} w_i T_g(t_i) \\ &= \sum_{i=1}^{k} w_i \left(e(c) + e'(c)(t_i - c) + e''(\xi_e)\frac{(t_i - c)^2}{2} \right) + \\ &\sum_{i=k+1}^{n} w_i \left(g(c) + g'(c)(t_i - c) + g''(\xi_g)\frac{(t_i - c)^2}{2} \right). \end{aligned} \tag{6.10}$$

This equation is not useful in its current form, as we would like to understand how this approximation behaves in terms of h. Using the assumption that $h = b - a = \sum_{j=1}^{n} w_j$ and the definition that $e(c) = g(c)$, the terms not involving derivatives can be rewritten as

$$Q_n^0 = \sum_{i=1}^{k} w_i e(c) + \sum_{i=k+1}^{n} w_i g(c) = \sum_{i=1}^{n} w_i e(c) = he(c). \tag{6.11}$$

The terms involving first derivatives are given by

$$Q_n^1 = e'(c) \sum_{i=1}^{k} w_i (t_i - c) + g'(c) \sum_{i=k+1}^{n} w_i (t_i - c). \tag{6.12}$$

In order to represent this equation in terms of h (i.e., the spacing between a and b), we note that

$$\begin{aligned} t_i &= a + \beta_i h \\ c &= a + \beta_c h \\ t_i - c &= a + \beta_i h - (a + \beta_c h) = h(\beta_i - \beta_c) \end{aligned} \tag{6.13}$$

where $0 \leq \beta_i \leq 1$. Substituting these relations into Equation 6.12, we get

$$Q_n^1 = e'(c) \sum_{i=1}^{k} w_i h(\beta_i - \beta_c) + g'(c) \sum_{i=k+1}^{n} w_i h(\beta_i - \beta_c). \tag{6.14}$$

Expanding the and rearranging the summation yields:

$$\begin{aligned} Q_n^1 = he'(c) \sum_{i=1}^{k} w_i \beta_i + hg'(c) \sum_{i=k+1}^{n} w_i \beta_i - \\ h\beta_c e'(c) \sum_{i=1}^{k} w_i - h\beta_c g'(c) \sum_{i=k+1}^{n} w_i. \end{aligned} \tag{6.15}$$

It is not immediately apparent how the individual weights are related to the interval spacing h. To elucidate the relationship, we first rescale the weights from $[a, b]$ to $[0, 1]$

$$w_i = \hat{w}_i(b - a) = h\hat{w}_i \tag{6.16}$$

where $\hat{w}_i$ is the rescaled weight. Substituting this relation into Equation 6.15 and simplifying the expression in terms of h yields:

$$\begin{aligned} Q_n^1 = h^2 \Bigg(e'(c) \sum_{i=1}^{k} \hat{w}_i \beta_i + g'(c) \sum_{i=k+1}^{n} \hat{w}_i \beta_i - \\ \beta_c e'(c) \sum_{i=1}^{k} \hat{w}_i - \beta_c g'(c) \sum_{i=k+1}^{n} \hat{w}_i \Bigg). \end{aligned} \tag{6.17}$$

A similar analysis reveals that the term involving the second derivative is $O(h^3)$. Therefore, Equation 6.10, written in terms of h, is

$$\begin{aligned} Q_n = he(c) + h^2 \Bigg(e'(c) \sum_{i=1}^{k} \hat{w}_i \beta_i + g'(c) \sum_{i=k+1}^{n} \hat{w}_i \beta_i - \\ \beta_c e'(c) \sum_{i=1}^{k} \hat{w}_i - \beta_c g'(c) \sum_{i=k+1}^{n} \hat{w}_i \Bigg) + O(h^3). \end{aligned} \tag{6.18}$$

The error is then found by subtracting Equation 6.18 from Equation 6.9, which gives the following expression for the remainder term:

$$E = \frac{(b-c)^2}{2} g'(c) - \frac{(a-c)^2}{2} e'(c) - Q_n^1 + O(h^3). \tag{6.19}$$

Since both $(b-c)^2$ and $(a-c)^2$ are $O(h^2)$, the error of an n-point quadrature rule, given our two aforementioned assumptions (i.e., the quadrature rule consists of $n \geq 2$ samples and the breakpoint occurs between sample points), exhibits worst-case second-order convergence.

If there are two breakpoints in the interval, a similar analysis indicates that an n-point quadrature rule, where $n \geq 3$ and in which the breakpoints lie intertwined between breakpoints, will exhibit worst-case linear convergence. For two breakpoints $c_0, c_1 \in (a, b)$:

$$f(t) = \begin{cases} f_0(t) & a \leq t \leq c_0 \\ f_1(t) & c_0 < t \leq c_1 \\ f_2(t) & c_1 < t \leq b. \end{cases} \tag{6.20}$$

Using Taylor's theorem as we did in Equation 6.7, we can rewrite $f(t)$ as

$$f(t) = \begin{cases} f_0(t) = T_0(t) = f_0(c_0) + f_0'(c_0)(t-c_0) + O(h^2) & a \leq t \leq c_0 \\ f_1(t) = T_1(t) = f_1(c_0) + f_1'(c_0)(t-c_0) + O(h^2) & c_0 < t \leq c_1 \\ f_2(t) = T_2(t) = f_2(c_1) + f_2'(c_1)(t-c_1) + O(h^2) & c_1 < t \leq b \end{cases} \tag{6.21}$$

and can then evaluate the integral as

$$\begin{gathered} \int_a^b f(t)\, dt = \int_a^{c_0} T_0(t)\, dt + \int_{c_0}^{c_1} T_1(t)\, dt + \int_{c_1}^{b} T_2(t)\, dt = \\ (c_0 - a) f_0(c_0) + (c_1 - c_0) f_1(c_0) + (b - c_1) f_2(c_1) + O(h^2). \end{gathered} \tag{6.22}$$

We next rewrite the quadrature formula, assuming that the breakpoints are between two sample points, $c_0 \in (t_j, t_{j+1})$ and $c_1 \in (t_k, t_{k+1})$, yielding:

$$Q_n = \sum_{i=1}^{j} w_i T_0(t_i) + \sum_{i=j+1}^{k} w_i T_1(t_i) + \sum_{i=k+1}^{n} w_i T_2(t_i). \tag{6.23}$$

The terms not involving derivatives are given by

$$Q_n^0 = \sum_{i=1}^{j} w_i f_0(t_i) + \sum_{i=j+1}^{k} w_i f_1(t_i) + \sum_{i=k+1}^{n} w_i f_2(t_i). \tag{6.24}$$

We can then use the relation in Equation 6.16 to express this in terms of h

$$Q_n^0 = h\sum_{i=1}^{j} \hat{w}_i f_0(t_i) + h\sum_{i=j+1}^{k} \hat{w}_i f_1(t_i) + h\sum_{i=k+1}^{n} \hat{w}_i f_2(t_i). \tag{6.25}$$

Subtracting Equation 6.24 from Equation 6.22, and using the fact that $f_0(c_0) = f_1(c_0)$, we get

$$\begin{aligned} E = {} & f_0(c_0)(c_1 - a) - h f_0(c_0) \sum_{i=1}^{k} \hat{w}_i \\ & + f_2(c_1)(b - c_1) - h f_2(c_1) \sum_{i=k+1}^{n} \hat{w}_i. \end{aligned} \tag{6.26}$$

Since $(c_1 - a)$ and $(b - c_1)$ are $O(h)$, the error of an n-point quadrature rule when there are two breakpoints in the domain exhibits worst case linear convergence. Similar analysis indicates that this result holds for $n > 2$ breakpoints as well.

We can see from the results in Equations 6.19 and 6.26 that the quadrature error exhibits worst case quadratic and linear convergence, respectively, under the assumption that the locations of the breakpoints do not coincide with any of the sample points, and that the quadrature rule uses two or more samples. While the assumption of two or more sample points covers many popular types of quadrature schemes (e.g., Newton-Cotes, Gaussian), it does not apply to two commonly used methods based on a single sample: Riemann quadrature and the midpoint rule. Neither of these rules will enable faster convergence; however, since, in the best case as applied to smooth functions, Riemann quadrature exhibits worst-case linear convergence and the midpoint rule exhibits worst case quadratic convergence.

6.3.2 Composite Quadrature of Piecewise-Smooth Segments

In practice, the volume rendering integral along a ray is evaluated by breaking up the domain into N subintervals and then applying an n-point quadrature rule to each subinterval. Consider the quadrature of a smooth function (i.e., there are no breakpoints) on $[A, B]$ where the size of each subinterval is specified as $h = (B-A)/N$. If we use a quadrature rule with a local truncation error of $O(h^n)$ for each interval, then the overall error of this composite rule is given by:

$$E = \sum_{i=1}^{N} O_i(h^n) = NO(h^n) = O(h^{n-1}) \tag{6.27}$$

where O_i represents the order of the error in the i^{th} subinterval. In contrast, consider a function that is not smooth and consists of N_1 intervals containing no breakpoints and N_2 intervals containing a single breakpoint. From the analysis above, we can see that the composite error will be

$$E = \sum_{i=1}^{N_1} O_i(h^n) + \sum_{i=1}^{N_2} O_i(h^m) \tag{6.28}$$

where $m = 2$ if the interval contains a single breakpoint, and $m = 1$ if it contains two or more. Therefore, the convergence of the entire integral is of order m.

The practical meaning of this result is that, when evaluating the volume rendering integral, if we can detect that a breakpoint exists along the ray, then it is sufficient to use the trapezoidal rule to evaluate the integral. Using methods designed for higher-order convergence will not produce faster convergence and may introduce additional error due to unnecessary floating point calculations.

To illustrate, we present two examples. In Figure 6.1, we show the convergence of several quadrature methods when applied to the function

$$f(t) = \begin{cases} -2t^3 + 4t^2 - t + .688528 & t \leq 0.54 \\ 2t^3 - 4t^2 + t + 1.311472 & t > 0.54 \end{cases} \tag{6.29}$$

which contains a single breakpoint at $t = 0.54$. Notice that Simpson's rule and Gauss-Legendre quadrature both exhibit second-order convergence, even though, in

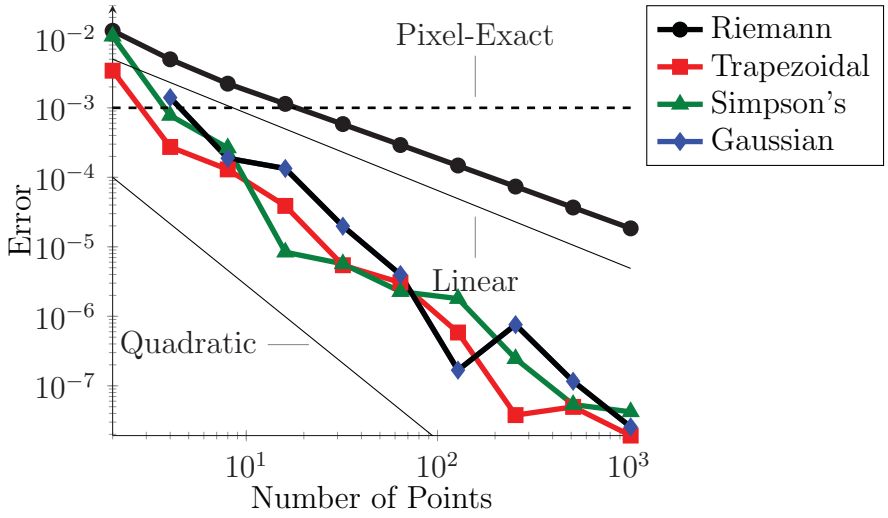


Figure 6.1. Convergence of composite quadrature methods when applied to a function containing a single breakpoint, using error as defined in Equation 6.6.

all intervals except the interval with the breakpoint, they approximate the integral exactly.

In Figure 6.2, we show the convergence of the volume rendering integral over a synthetic field (described in Section 6.6) where each ray consists of a transfer function with two breakpoints with a spacing of $w = 0.2, 0.02, 0.002$ between them. What we see is that the image converges linearly while the sample spacing is large enough to contain at least two breakpoints. Once the sampling falls below that threshold, the anticipated second-order convergence is observed.

6.3.3 Quadrature of Smooth Segments

When the transfer function along a ray is smooth (i.e., does not contain any breakpoints), we can take advantage of the structure of the high-order field to use high-order quadrature to evaluate the volume rendering integral. Since the field is defined by high-order polynomials in reference space, a natural choice of integration method is Gaussian quadrature, which can exactly integrate a $2n-1$ order polynomial with n function evaluations. Since the field along the ray is only guaranteed to be smooth, Gaussian quadrature will be unable to evaluate the integral exactly. The resulting error can be reduced by subdividing the ray and using Gaussian quadrature

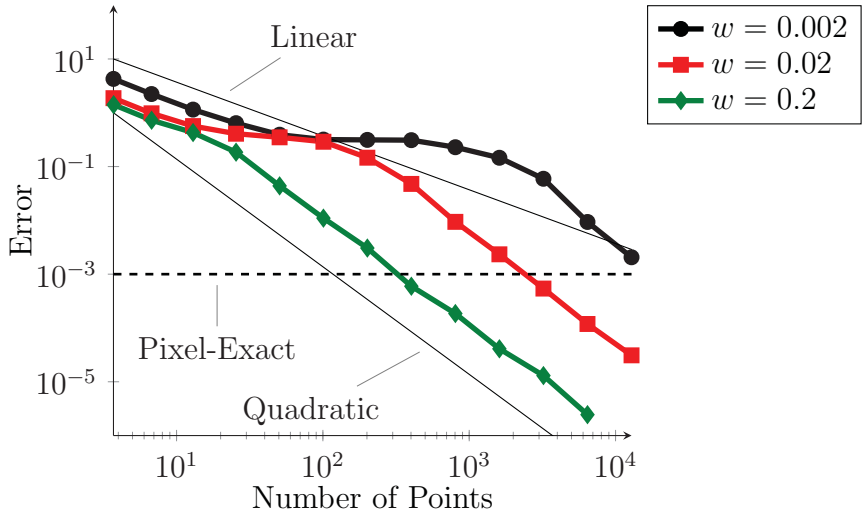


Figure 6.2. Convergence rates for transfer functions with closely-spaced breakpoints (spacing indicated by w). Second-order convergence is only possible when every quadrature interval contains at most one breakpoint. The closer the breakpoints, the longer it takes to converge to a pixel-exact image.

on each subinterval. The disadvantage of this approach is that the points from the subintervals do not coincide with the original evaluation points, requiring additional samples at all points each time a subdivision is performed.

Since we would like high-order convergence combined with the reuse of sample points, we turn to Gauss-Kronrod quadrature [29]. This method consists of an n point Gaussian quadrature estimate, followed by an $n+1$ point Kronrod extension, for a total of $2n+1$ function evaluations. This rule is exact for polynomials up to degree $3n+1$. Since the field along the ray is smooth, we can obtain an error estimate by looking at the difference between the n point Gaussian rule and the $2n+1$ Kronrod extension.

6.4 Evaluation of the Volume Rendering Integral

We now discuss how we apply the concepts from the previous section to evaluate the volume rendering integral along a ray.

6.4.1 Empty Space Skipping

Empty space skipping is an important acceleration technique that has shown considerable performance improvements on data sets where sampling is cheap, such as voxel-based volumes where sampling involves trilinear interpolation. Space skipping is even more important in the context of high-order volume rendering because sampling the field is considerably more expensive. Recall from Section 6.1 that sampling a point requires the numerical inversion of the mapping function Φ as well as the evaluation of a high-order polynomial. Therefore, performance improvements can be realized by accurately detecting segments along the ray that do not contribute to the volume rendering integral.

Empty space skipping is performed on a ray segment in which τ is zero along the entire segment, which indicates that the segment does not contribute to the volume rendering integral. The performance implications of this optimization are dependent on the nature of the high-order field and of the transfer function. Transfer functions that classify large portions of the range of the scalar field will not need to skip many sections, while transfer functions that classify only targeted segments of the field will see performance improvements.

6.4.2 Occlusion Only

We evaluate occlusion only when the density transfer function has a value ($\tau \in \mathbf{PS} \vee \tau \in \mathbf{S}$) and the color transfer function is zero along the ray ($\kappa \in \mathbf{E}$). In this case, we do not need to evaluate the outer integral, as there is no emissive component to the volume rendering integral. We do, however, need to evaluate the accumulated opacity along this segment. We do this by using the trapezoidal rule if the transfer function is piecewise-smooth, and Gauss-Kronrod quadrature if it is smooth, as discussed in Sections 6.3.2 and 6.3.3.

6.4.3 Evaluating the Outer Integral

For all remaining cases, we must evaluate both the inner and outer integrals, as both the density and color transfer functions contribute to the final color. The convergence rate of the outer integral depends on the type of quadrature chosen for the outer and inner integrals. To see why this is the case, let Q_n be the quadrature

rule used for the outer integral with error $O(h^n)$, and I_m be the quadrature rule used for the inner integral, with error $O(h^m)$. Evaluating the outer integral using Q_n gives:

$$\int_a^b f(t)\,dt = \sum_i \left(w_i f(t_i) + O(h^m)\right) + O(h^n). \tag{6.30}$$

The term $O(h^m)$ in the summation comes from evaluating the inner integral on $[a, t_i]$ using quadrature rule I_m. This constrains the convergence of the outer integral to be no greater than the convergence of the inner integral.

If either κ or τ are piecewise-smooth, then the entire outer integral will have either second- or first-order convergence, depending on the location of the breakpoints. Therefore, even if one of κ or τ is smooth, we use the trapezoidal rule to evaluate both the outer and inner integrals. This has the added benefit of using the same set of sampling points for both integrals, reducing the overall number of computations required.

If κ and τ are both smooth, then we can obtain high-order convergence by applying Gaussian quadrature to both the outer and inner integrals. In practice, however, this does not work very well, since the evaluation points of the outer integral do not, in general, correspond to the points required to evaluate the inner integral. So, for each sample t_i in the outer integral, we would need to resample all points to evaluate the inner integral, which negates the anticipated performance improvements of high-order quadrature. We therefore evaluate the outer integral using Gaussian quadrature, but evaluate the inner integral using the trapezoidal rule. While this limits us to first- or second-order convergence, we show in Section 6.6.1 that it does give us better accuracy for a given number of samples.

6.4.4 Adaptive Quadrature

From the analysis presented above, we cannot expect to achieve better than second-order convergence when evaluating the volume rendering integral. A natural next step is to consider using adaptive quadrature to both reduce the number of samples required to evaluate the integral, and to generate pixel-exact images through the use of error estimators. This approach is especially appealing considering that

transfer functions are often designed to ignore portions of the scalar field to allow the user to focus on features of interest. Transfer functions that do this contain one or more segments, $\tau(s) = 0, s \in [s_{min}, s_{max}]$, where the transfer function does not contribute to the integral's result. By using adaptive quadrature, we hope to avoid sampling these areas, and instead concentrate our samples on the portions of the transfer function that do contribute to the result.

Using adaptive quadrature, pixel-exact images are generated by refining the integral until the error estimate falls below the tolerance required for pixel-exact images. Since there are 256 color levels for each channel in a standard 24-bit color image, we consider each color channel to be pixel-exact once the estimated error in the channel's integration falls below $1/256 = 0.0039$. To provide a buffer against underestimation of the error, we use 0.001 as our threshold for a pixel-exact image. Therefore, when an adaptive volume rendering has been performed, the resulting image can be known to be pixel-exact and, because we were able to sample adaptively, we are also able to reduce the number of samples required to generate the image.

We attempted several implementations of adaptive trapezoidal quadrature on the GPU and, in each case, we found that we were able to reduce the number of samples required to evaluate the integral and generate pixel-exact images. This came, however, at the cost of an overall increase in image generation time. In some cases, execution time doubled when using adaptive quadrature compared to nonadaptive quadrature, even though adaptive quadrature reduced the number of samples required. This performance result can be attributed to the GPU architecture we are using. When using Cuda for sampling the field, the simple trapezoidal rule with constant spacing h between samples can be implemented efficiently by first loading the basis functions into memory, then evaluating each of the samples in parallel. With adaptive quadrature, however, we interfere with the GPUs ability to evaluate the samples in parallel. At each step in an adaptive quadrature algorithm, we must determine if the current segment of the ray meets our predetermined accuracy requirements and, if it does not, subdivide the segment in a recursive manner until it does. In this manner, even though we may be evaluating fewer samples overall, our GPU is not working as efficiently as it could, and the overall execution time is slower.

At this time we have not found a way to perform adaptive quadrature on the GPU in a way that improves performance. The methods we have tried are able to reduce the number of samples taken at the expense of increased execution time, which defeats the purpose of reducing the number of samples required. So while adaptive quadrature is an attractive approach in principle, we have not yet found a feasible implementation.

6.5 Implementation

The implementation of the algorithm shown in Algorithm 2 is done using a combination of OptiX and Cuda. The OptiX ray tracer is responsible for traversing the volume on an element by element basis. During each iteration, it stores the current element and the entrance and exit points $[t_a, t_b]$ for the element along the ray. Our initial implementations evaluated the volume rendering integral in OptiX as well. This was problematic for two reasons. First, the code necessary to perform the ray-tracing is compiled by the OptiX engine at runtime. The code for evaluating the volume rendering integral resulted in runtime initialization times on the order of 10 to 20 minutes. Secondly, and more importantly, evaluating the volume rendering integral in OptiX prevented us from using the GPU to its fullest extent. In particular, we have been able to achieve better performance by using shared memory, which is unavailable for use within an OptiX kernel.

Once the ray tracer has completed, a Cuda kernel is launched to evaluate the volume rendering integral on each ray segment. The kernel needs to be able to access the data stored by the OptiX program, but the current Cuda implementations do not allow for the direct sharing of memory between Cuda contexts (OptiX is built on a Cuda context). While memory can be copied from OptiX to main memory, and them from main memory to the Cuda integration context, this is far too expensive for our interactivity requirements. Therefore, to share data between the OptiX and Cuda contexts, we create OpenGL pixel buffer objects to store the intersection points and element information. This approach requires the active GPU to be connected to a display to achieve best performance. If the active GPU is not connected to a display, such as is the case for GPUs used primarily for their computational capabilities, then

the pixel buffer object will be allocated on a GPU connected to a display. When the pixel buffer is used in either the OptiX or Cuda context, it is first copied to the active GPU, resulting in a significant performance loss.

The Cuda module then evaluates the volume rendering integral as described in Section 6.2. The loop continues until all rays exit the volume.

6.6 Results

We illustrate the utility of our algorithm, both in terms of the accuracy of the generated images and its performance characteristics, by using it on three different data sets. The first data set consists of a single, axis-aligned hexahedron with extents $[-1,-1,-2]-[1,1,4]$ and spherical field $f(x,y,z) = x^2+y^2+z^2$. This data set provides a good baseline upon which we can evaluate the accuracy of our implementation. The other data sets are the flow scenarios presented in Section 5.2.

All tests were performed on a desktop workstation equipped with an NVIDIA Tesla C2050 GPU and Intel Xeon W3520 quad-core processor running at 2.6 GHz. We used OptiX Version 2.1.1 and Cuda Version 4.0, using 32-bit floating point precision for all code executed on the GPU. Performance when using 64-bit precision varies depending on the card.

6.6.1 Convergence Results

The goal of our system is the generation of pixel-exact images. The convergence graphs shown in this section were obtained by first generating a pixel-exact image, then comparing this image to the images created with varying values of sample spacing. The error metric we used is the largest difference between pixel values over the image: $\text{Max}|I_r(x,y) - I_h(x,y)|$, where $I_r(x,y)$ is the pixel in the pixel-exact image at position (x,y), and $I_h(x,y)$ is the pixel in the image generated with sample spacing h. We compare this overall image error to the total number of samples required to generate the image.

We start by verifying the theoretical convergence of our method by using the spherical synthetic data. We use this data set because we can evaluate the volume rendering integral accurately enough to guarantee a pixel-exact image, which we can then use as a "gold-standard" against which to verify our method. Even for a field

as simple as this one we are unable to create an analytic expression for the volume rendering integral. However, by placing the field in an axis-aligned hexahedron, we know that the mapping function Φ from Section 6.1 is a linear transformation, which means that the field along the ray will be quadratic.

Generation of the pixel-exact image proceeds as follows. We start by generating a ray for each pixel. In this special case where we know the field along the ray is quadratic, we can numerically find the location of each breakpoint along the ray to obtain a list of ray segments with no discontinuities (similar to what is done by Williams et al. [64]). We then evaluate the outer integral using Gauss-Kronrod quadrature. At each of the sample points in the outer integral, we use Simpson's rule to evaluate the inner integral exactly. After evaluating all of the points in the outer integral, we use the Gauss-Kronrod error estimator to determine if we have achieved a pixel-exact image. We have found that for this simple data set, the 15-point Gauss-Kronrod rule approximated the volume rendering integral well below the pixel-exact error threshold.

We then generated visualizations using Riemann integration (which is representative of the types of integration performed by most existing volume rendering systems) and our system to verify the expected convergence rates.

Results are shown in Figure 6.3(a). As expected, Riemann quadrature converges linearly, while our method exhibits second-order convergence. Not only does our method exhibit higher-order convergence than Riemann quadrature, the resulting image is also more accurate for a given number of samples.

We next test our convergence on the canister data set for which we cannot calculate the volume rendering integral exactly. Since we do not have an analytic solution, we create the pixel image by refining the sampling size until the resulting image no longer changes. We show convergence results for both Riemann quadrature and our method in Figure 6.3. We can see that we achieved the expected second-order convergence, and that our method always returns an image with less error than the image produced using Riemann integration.

Finally, we illustrate the benefit of using high-order quadrature for the outer integral when possible. We applied a transfer function to the sphere data set in which

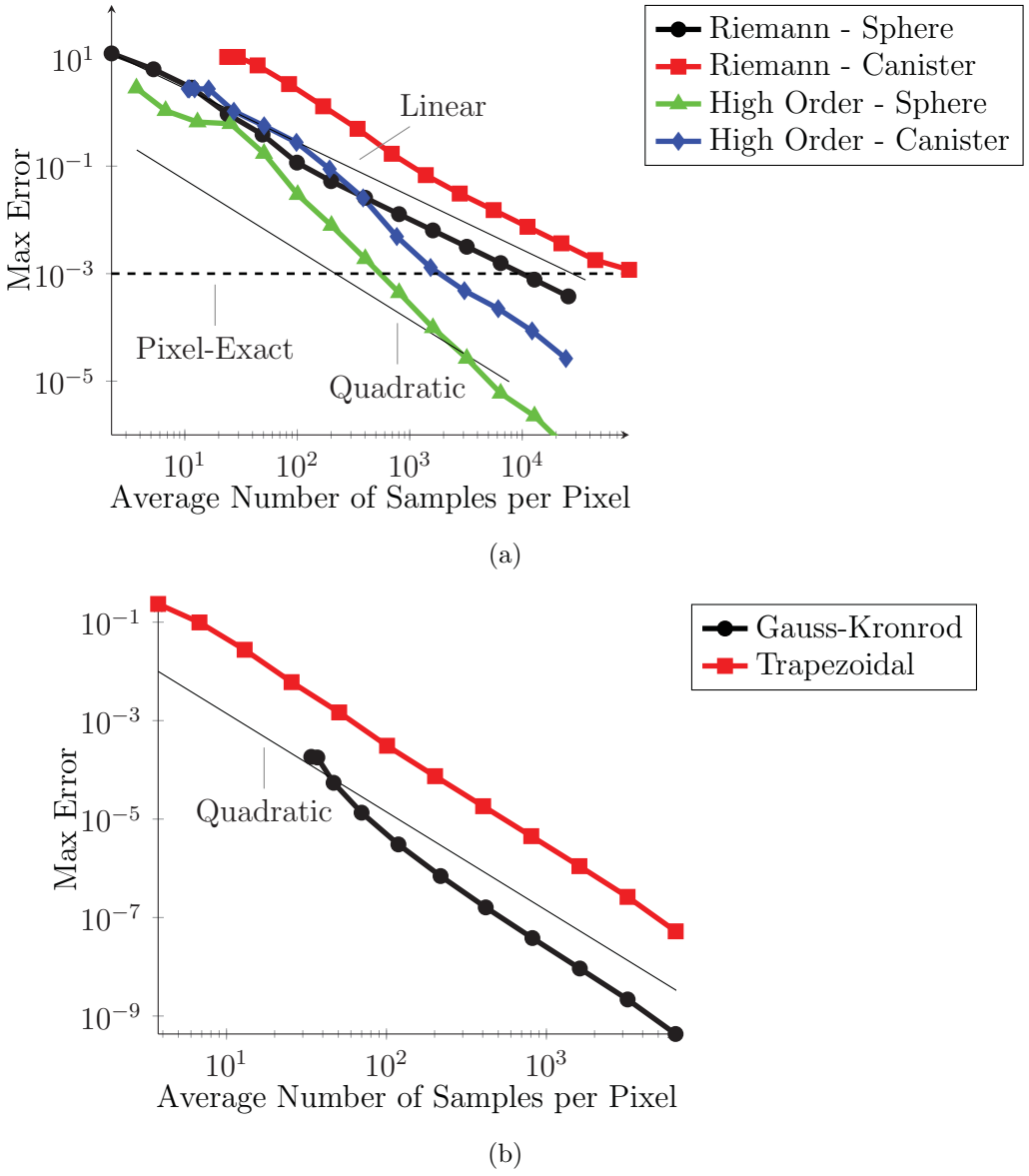


Figure 6.3. Convergence rates for different integration schemes. (a) - Comparison of convergence rates between Riemann integration and our high-order method. Our method converges to the pixel-exact image an order of magnitude faster than simple Riemann integration. (b) - Comparison of using Gauss-Kronrod quadrature to evaluate the outer integral of the sphere data set versus trapezoidal rule.

neither the density function nor the color transfer function contained breakpoints. As discussed in Section 6.4.3, we expect second-order convergence in this case. In Figure 6.3(b), we compare evaluating the volume rendering integral using Gauss-Kronrod quadrature and the trapezoidal rule for the outer integral. As expected, both methods converge quadratically, but by using Gauss-Kronrod quadrature for the outer integral, we are able to obtain a more accurate image for a given number of samples. In fact, for a given level of accuracy, we see that the Gauss-Kronrod approach uses an order of magnitude fewer samples.

6.6.2 Visual Comparisons

To understand the practical impact of our algorithm, we must not only understand the improved convergence rates discussed in the previous section, but also see if it is capable of producing a noticeably better image than existing algorithms when given the same amount of time with which to generate the image. To illustrate the effect our algorithm has on image generation, we created two images from each of the data sets described above: the canister (Figure 6.4) and the block and splitter plates (Figure 6.5). In each test, the image generated by our algorithm is compared to the image generated by evaluating the volume rendering integral using Riemann integration with evenly spaced samples. The transfer function for all tests is designed to display three semi-transparent isosurfaces.

Observe that, in both tests, our method is able to produce noticeably better images for a given number of samples and execution time. The images produced by Riemann integration suffer from banding and other artifacts associated with insufficient sampling, artifacts that do not appear in the images generated by our method.

6.6.3 Performance

While the primary focus of our system is on accuracy, we posit that it must also be interactive to be useful. We define interactivity as achieving rendering speeds of at least one frame per second. Overall, our system is capable of achieving these interactive frame rates for images of up to 512×512 pixels. For images larger than this, our system reduces the sampling rate during user interaction to maintain interactivity, and then generates the full, accurate image when user interaction has stopped.

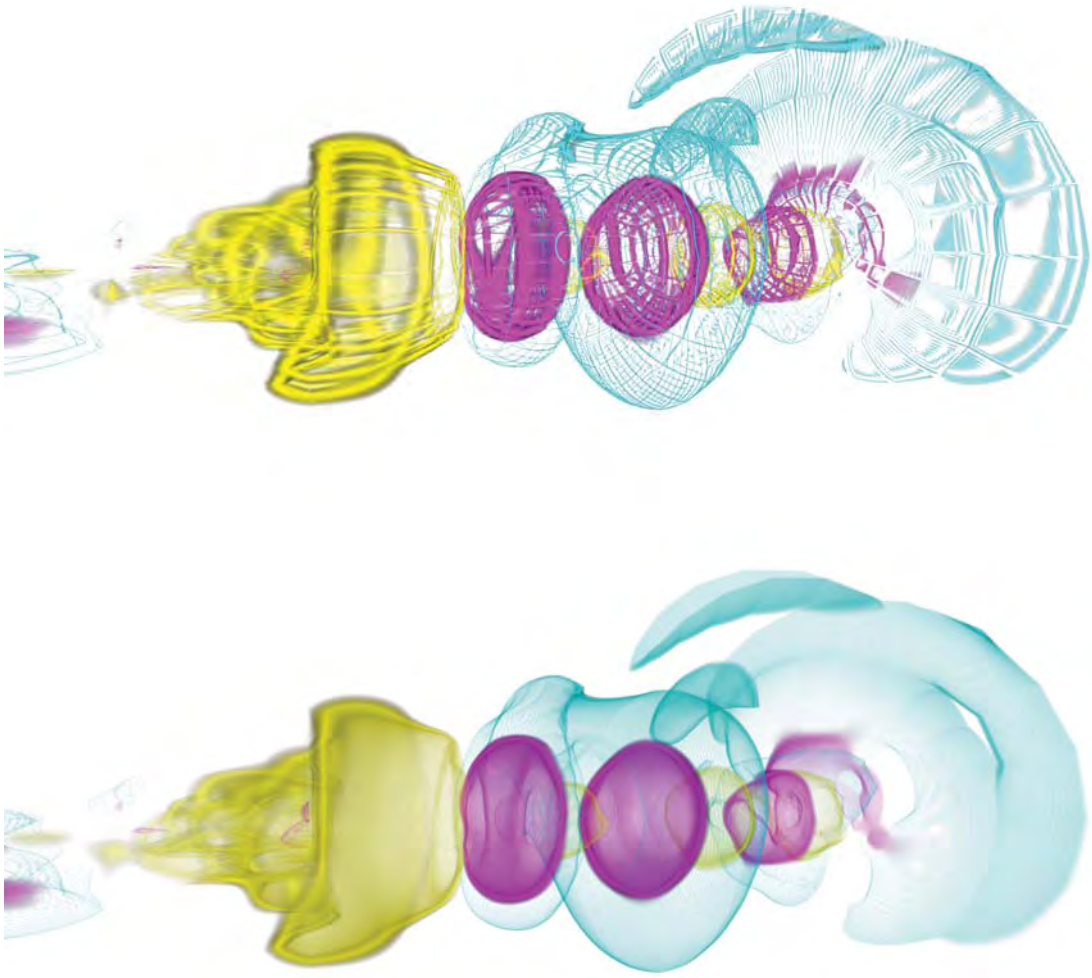


Figure 6.4. Volume rendering of the pressure of a high-order finite element solution of fluid flow past a rotating canister. (Top) - Image generated using Riemann integration and 60 million equally-spaced samples. (Bottom) - Image with improved accuracy generated using the algorithm presented in this paper using the same number of samples. Both images rendered in three seconds for a 1720×860 image.

Figure 6.5. Image quality comparison between Riemann sum evaluation (top) and our algorithm (bottom) for the block and splitter plates data set. Both images were generated using 140 million samples and rendered in two seconds for a 1600×800 image. Notice the banding present in the top image, which is an indication of insufficient sampling resolution.

The execution speed of our system is influenced by several factors: the desired sample spacing h along a ray, the overall image size, the polynomial order of the high-order data set, and the number of elements in the data set. To investigate the impact of each of these factors, we have performed experiments where we varied one of these parameters while holding the others constant.

In Figure 6.6, we show the performance of our system based on the number of samples used to evaluate the volume rendering integral. Performance scales linearly with the number of samples. Combined with the convergence analysis from Section 6.6.1, we see that we can generally expect to double execution time to reduce the error by four. In Figure 6.7, we show performance as a function of image size. Finally, in Figure 6.8, we show performance based on the polynomial order associated with each direction. While the time required does grow quickly with order, in practice volumes are rarely higher than $6^{th} - 8^{th}$ order per direction.

Of these parameters, we have control over h and the image size, but we do not have control over the number of elements or their order, as these are established by the engineer creating the simulation and are domain specific.

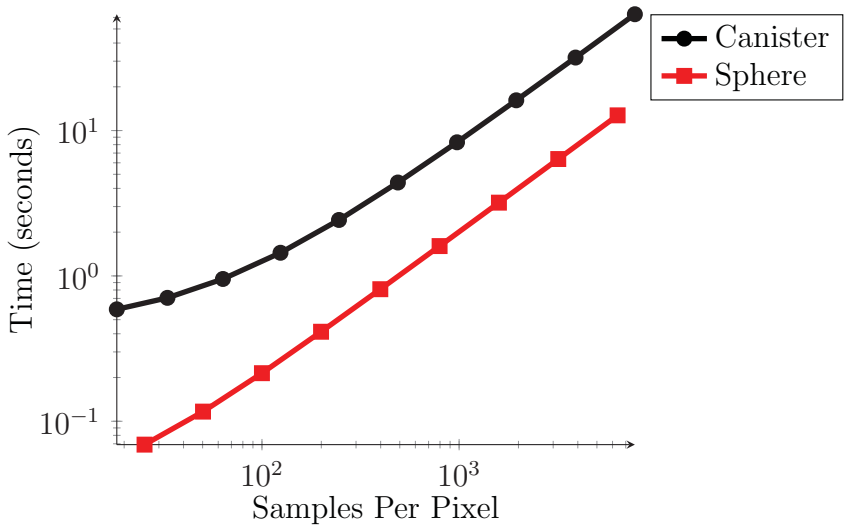


Figure 6.6. Time in seconds to render volumes based on number of samples required per pixel.

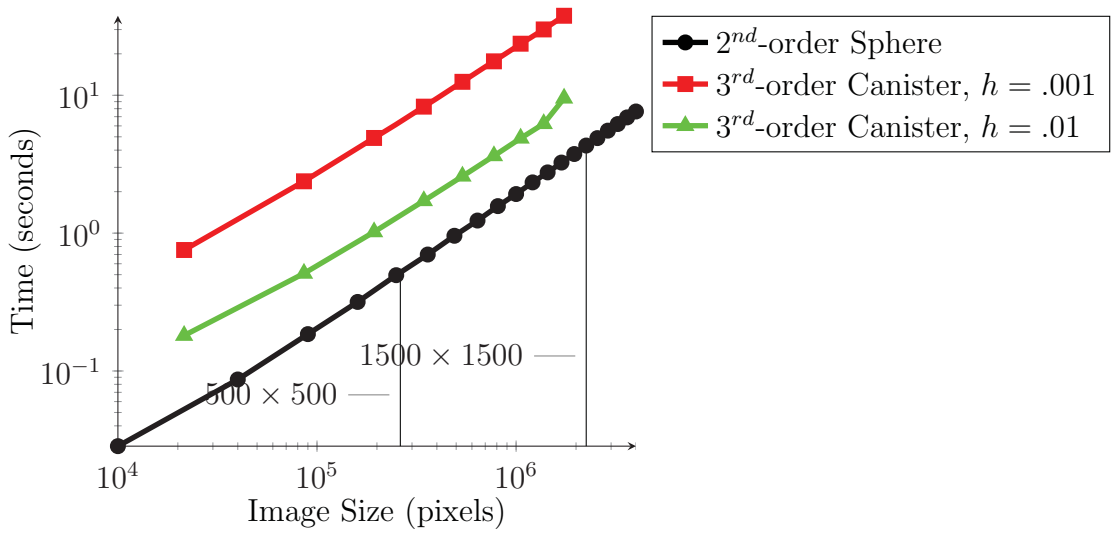


Figure 6.7. Time in seconds to render volumes based on image size.

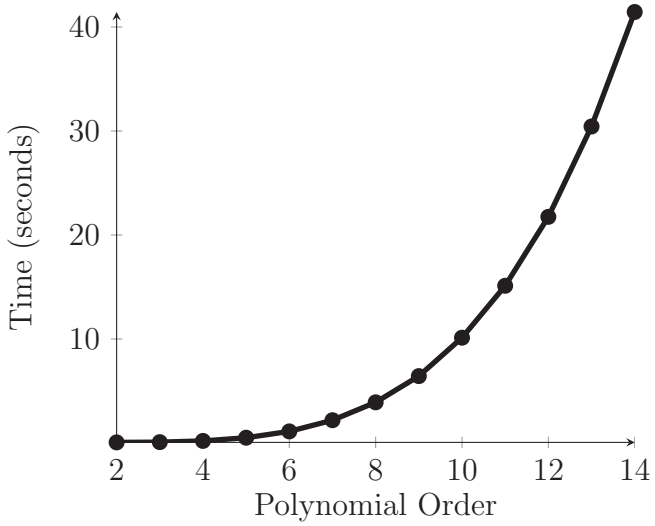


Figure 6.8. Time to render a 1120×1120 image with increasing polynomial order.

6.7 Summary

We have presented a new technique for evaluating the volume rendering integral in high-order finite element fields which addresses the often contradictory goals of image accuracy and interactive performance. This system uses the high-order data in its native form, thereby avoiding the approximation errors that are present when sampling onto voxel-based data structures. We have shown that, while the worst case convergence of our system is the same as that of simple Riemann integration, we generally achieve second-order convergence and are capable of producing images with less error for a given number of samples when compared to existing methods. By reducing the number of samples used to generate accurate images, we have been able to develop a system that can produce volume rendering images of high-order data efficiently on a desktop system.

Although our algorithm is capable of generating pixel-exact images via adaptive integration, we found that technical limitations made it far from interactive and unsuitable for general use. We are currently investigating new ways in which adaptive integration of the volume rendering integral can be framed in the context of GPU computation to restore the lost performance.

CHAPTER 7

THE ELEMENT VISUALIZER (ELVIS)

To address the accuracy and performance issues raised in the preceding chapters, we need an integrated visualization system that is designed specifically for high-order finite element solutions. Specifically, such a system must have the following features:

- **Extensible Architecture:** To support data originating from any high-order simulation, the system's visualization algorithms should be decoupled from the data representation, allowing them to change independently of each other. The advantage of this approach is that the visualization algorithms can be improved as new techniques and algorithms are developed, while engineers are free to choose whatever basis functions are most appropriate for the scenarios under investigation. This architecture enables the system to support methods currently in use, as well as methods that have not yet been developed.

- **Accurate Visualization:** To avoid introducing error into the visualization, the high-order system must work with the high-order data directly. Specifically, the system must be able to evaluate the solution at arbitrary locations in the domain to machine precision. The system must also support visualization methods that have been developed based on the a priori knowledge that the data was produced by a high-order finite element simulation. These methods will ideally make use of the smoothness properties of the high-order field on the interior of each elements, while respecting the breaks in continuity that may occur at element boundaries.

- **Interactive Performance:** In terms of computational resources required, using the high-order data directly carries significantly higher costs than using simpler linear approximations. While a high-order system is not expected to

provide the same level of performance as its linear counterparts, it should provide an interactive experience on a standard desktop workstation (i.e., it should not require expensive, special purpose hardware).

In this chapter we describe the Element Visualizer (ElVis), a new high-order finite element visualization system that meets the requirements listed above. We demonstrate ElVis' utility by using it to visualize finite element simulations produced by ProjectX, which is a general-purpose PDE solver with an emphasis on aerospace applications [15, 44, 10, 68, 67] developed at the Department of Aeronautics and Astronautics at MIT. Specifically, we will consider the visualizations necessary during the debugging and verification processes of model generation.

7.1 Overivew

ElVis is designed to provide visualization tools that are broadly applicable to any high-order finite element solution. ElVis' implementation is generic and aims to decouple the implementation of the visualization from the implementation of the high-order basis functions. ElVis achieves this goal through the use of plugins, which provide a simplified interface to the high-order data by exposing the minimal amount of functionality required to generate a visualization. In this way, it is broadly applicable to a wide variety of simulation products, and gives each product wide latitude on how it behaves behind the scenes. We discuss plugins in more detail in Section 7.2.

Once the data is accessible to ElVis through a plugin, ElVis can perform the required visualizations without knowledge of the details of the underlying simulation. ElVis' visualization algorithms focus particular attention on the two often competing goals of image accuracy and interactive performance. Image accuracy is obtained by devising high-order specific versions of common visualization strategies (cut-surfaces, isosurfaces, and volume rendering). Performance is achieved by careful implementation of these algorithms as parallel algorithms on a NVIDIA GPU, using the OptiX [45] ray-tracing engine and Cuda [2] as the framework. We present more details about ElVis' visualization capabilities in Section 7.3.

7.2 Extensibility Module

One of the fundamental challenges of creating a general-purpose visualization system for high-order finite element simulations is that there is no single set of basis functions that is appropriate in all simulation settings. Therefore, each simulation system chooses the basis that is most suited for the problems at hand. This means that ElVis cannot be implemented in terms of any specific basis and expect to used with arbitrary simulation systems. One way this can be addressed is to require users of each simulation system to first convert their data into ElVis' native file format, which is a tedious, error-prone process that requires significant understanding about the details of the target format.

The Extensibility Module addresses these issues by providing a plugin interface that acts as a bridge between the visualization system and the simulation package. The module accepts plugins written in one of two ways, each providing different trade-offs as described below. The first type of plugin is the *runtime plugin*, which provides an interface for ElVis to interactively query the simulation data on both the CPU and GPU. The second plugin type is the *data conversion plugin*, which is used to convert a data set from the format used by the simulation package to the format used by ElVis' default plugin, the Nektar++ extension [3].

The advantage of a runtime plugin is that ElVis is able to operate directly on the high-order data, without needing to resort to an intermediate representation. The downside is that it can require a significant amount of coding, especially for those cases where the basis functions are not implemented in a C-like language. This type of program also requires some familiarity with GPU programming: OptiX and Cuda in particular.

Data conversion plugins have the advantage of requiring far less code than a runtime plugin. In most cases, the only necessary functionality is translating the data requested by ElVis into a form understood by the simulation. This type of plugin must be written in C++, but does not require any GPU programming. The downside is that this plugin will introduce error if the simulation's data cannot be represented with polynomials.

7.2.1 Data Conversion

The purpose of the data conversion plugin is to convert fields and geometry from the format used by the simulation package into the native Nektar++ format used by ElVis. The Nektar++ data format is supported through a default runtime plugin (described below) that is distributed with ElVis as a reference implementation for the development of plugins for other simulation systems. Nektar++ uses a polynomial basis to represent its data. Therefore, if the simulation's basis is also expressed in terms of polynomials, even polynomials of a different form, no projection error, beyond that of floating-point rounding error, is introduced in the conversion.

The data conversion plugin requires two core inputs: a collection of elements (as described in Section 3.1), and one or more scalar fields. The bridge between ElVis and user code is contained in a collection of methods designed to obtain information about the data set and evaluate the scalar fields at arbitrary points in the domain. It is anticipated that, in implementing these methods, programmers will be responsible for translating from ElVis' API to the simulation's data representation, and will not need to reimplement any existing functionality. It is this feature that provides one of the primary advantages of the data-conversion plugin—the minimal amount of code required to implement it.

Projection of the data then occurs as follows. First, ElVis queries the plugin to obtain information about the each element's type (e.g., hexaheron, tetrahedron) and the desired polynomial order of the converted data set. For simulations already using a polynomial basis, this can be chosen so that the projection introduces no error beyond floating-point rounding errors. For other bases, it can be set to the level needed to meet the desired accuracy requirements. ElVis then queries the plugin to determine if the resulting projection should be represented using functions that are continuous or discontinuous at element boundaries. Finally, ElVis samples the field at a collection of points determined by the choices made in the previous steps and creates the projected data set.

The advantages of this approach when compared to runtime plugins (described below) are that they will generally require less coding and, once the conversion is done, ElVis will have no runtime dependencies on the simulation package. Another

advantage is that ElVis handles all of the details about file formats and data storage—the plugin is only responsible for sampling the solution. The downside is that the native internal format represents fields and geometry as the tensor product of one-dimensional polynomials [24]. Therefore, data sets from simulation packages where fields are represented by nonpolynomial basis functions cannot be represented exactly, so this approach will introduce projection errors into the visualization. Another disadvantage is that simulations using nonstandard elements do not fit the input requirements described above and cannot be converted.

7.2.2 Runtime Plugins

Runtime plugins are loaded into ElVis each time it is run; they provide access to a simulation's data during the rendering process. The data can be accessed directly in the format used by the simulation without the need to convert formats first. However, implementing a runtime plugin requires significantly more code than the data conversion approach, and it also requires a working knowledge of both OptiX and Cuda. All of ElVis' visualization algorithms are implemented on the GPU; therefore, all runtime plugins must provide a means to access data fields on the GPU.

A runtime plugin consists of three components:

- **Volume Representation Component:** This component is responsible for reading a volume on the CPU and then transferring it to the GPU. ElVis imposes only one restriction on the way in which the volume's data is represented and accessed on the GPU, leaving the choice of optimal implementation to the extension. The sole requirement is that the data be accessible to the OptiX-based ray tracer through a specially named node in the ray tracer's scene graph.

- **Volume Evaluation Component:** All of the visualization methods described in the next section require the ability to evaluate a high-order field and its gradient at arbitrary locations within an element. These functions must be implemented on a GPU and will be called a large number of times for each of the visualization methods discussed in the next section, so extra care must be taken to ensure that they are as efficient as possible.

- **Ray-Tracing Component:** Finally, the ray-tracing component connects the OptiX portion of ElVis to the simulation data. This component is not responsible for handling the high-level mechanics of the ray tracer; however, it is responsible for providing some of the primitives used by ElVis to perform the ray-tracing. Examples include providing ray-element intersection tests, element and element face bounding box procedures, and tests for whether a point is located in an element.

7.3 Supported Visualizations

In this section, we discuss the visualization methods that are available in ElVis.

7.3.1 Surface Visualization

ElVis supports the rendering of an arbitrary number of cut-planes and surfaces, of which the curved faces lying on domain boundaries are common choices. In Figure 7.1, we show an example of the pressure field on an ONERA M6 wing (see Section 7.4.3 for details) and on a plane cutting through the wing. A color map is applied and contour lines are plotted on the wing's curved surface and on the cut-plane's flat surface.

ElVis also has the ability to plot the intersection of the 3D mesh and a surface through an extension of the contouring algorithm discussed above. It is often useful to see the mesh on a surface to verify that the mesh has been generated correctly,

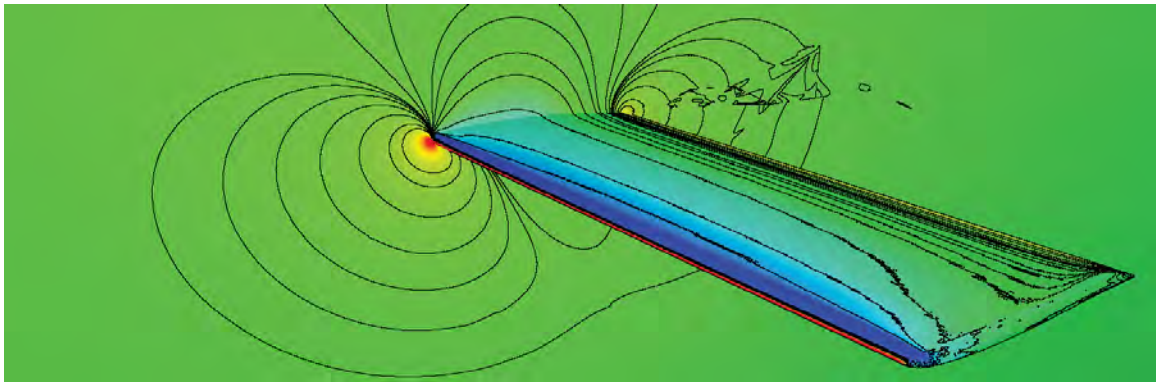


Figure 7.1. Density field on the ONERA M6 wing (Section 7.4.3), rendered using ElVis, illustrating the application of color maps and contours on curved and planar surfaces.

as well as to aid in debugging. Oftentimes features may appear in the visualization that appear out of place, but turn out to be reasonable if they occur next to a mesh boundary. An example of such a feature is a discontinuous contour line in a DG field. In this scenario, a break in the contour is expected at element boundaries, but not in the interior of the element. Further examples that apply the meshing tool can be found in Section 7.4.

7.3.2 Isosurfaces

An advantage this method has over existing object-space methods is that it respects the features of the high-order data. In particular, unless care is taken, object-space isosurfacing methods such as marching cubes can miss valid features of DG simulations, such as discontinuities across element boundaries that can cause cracks in the isosurface, and isosurfaces that exist entirely within an element. An example of this can be seen in Figure 7.2, where we plot an isosurface of the Mach number for the delta-wing simulation described in Section 7.4.3.

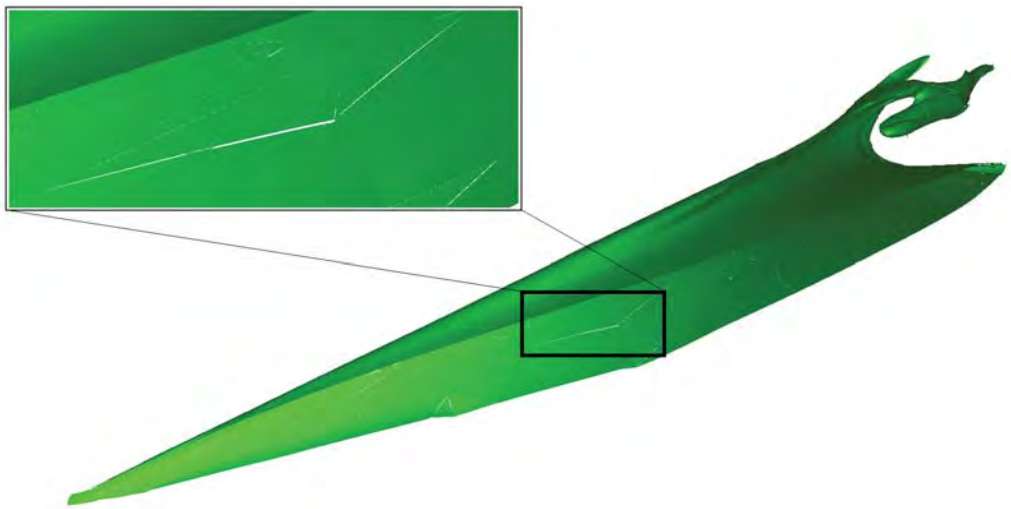


Figure 7.2. Isosurface of Mach number 0.1919 for the delta wing simulation (Section 7.4.3), showing the development and roll up of the vortex structures along the leading edge and downstream of the wing. Note the crack in the surface arises because the underlying solution is from a DG method.

7.3.3 Combining Visualizations

A constant challenge for high-order visualization is the computational cost of each of the algorithms described above. This comes from the cost of sampling a high-order field (primarily the cost of converting world points to reference points and evaluating the field) and the cost of traversing the volume with a ray (where ray-element intersections can be of high-order and require expensive, iterative root finding procedures). Several of the visualization algorithms above have been shown to be interactive while maintaining accuracy when applied in isolation; but combining these visualization methods into a single image without taking into account their interactions leads to slow performance.

Instead, when multiple visualization methods are used in a single image, we setup a system of communication and sharing between modules to minimize the cost. Some optimizations include:

- **Sample sharing:** When a ray intersects a surface and takes a sample, we use the information from that sample to draw the mesh and generate color maps and contours without the need for additional rays.

- **Occlusion sharing:** Of the algorithms listed above, volume rendering and isosurface generation are significantly more expensive than the rendering of surfaces. Therefore, we render the surfaces first and obtain depth values for each pixel corresponding to the location of the intersection. During isosurface generation and volume rendering, we terminate the mesh traversal portion of the algorithms once we have reached the occluding structure.

- **Mesh traversal sharing:** Mesh traversal is an expensive operation, especially when curved element faces are involved. To aid performance during both isosurface and volume rendering, each ray moves from element to element, evaluating the isosurface and volume rendering integral for each segment before moving on to the next segment.

7.4 Results

This section demonstrates the capabilities and advantages of ElVis through several examples. Our examples are taken largely from engineering problems solved or being worked on with ProjectX; some examples were toy problems designed to demonstrate specific capabilities. The following subsection describes the sources of our examples, enumerated as series of cases. Then we show results and comparisons with current visualization "best-practices" used by ProjectX developers.

7.4.1 ProjectX

All results generated in this section are from solutions produced by the ProjectX software. The results considered in this paper arise from solutions of the compressible Navier-Stokes and RANS equations. ProjectX implements a high-order DG finite element method; DG features relevant to visualization were discussed in Section 3.1. It also strives to increase the level of solution automation in modern CFD by taking the engineer "out of the loop" through estimation and control of errors in outputs of interest (e.g., lift, drag) [15, 68]. Solution automation is accomplished through an iterative mesh optimization process, which is driven by error estimates based on the adjoints of outputs of interest [67].

We have worked closely with ProjectX engineers to endow ElVis with visualization and interface features that they would find useful. Visualization has great potential to aid ProjectX developers' ability to understand and analyze their solutions. It has perhaps even greater application in the realm of software debugging, where visual accuracy is of the utmost importance since it is often difficult to discern visual artifacts from genuine or erroneous (i.e., a result of a software bug) solution features. As we will discuss below, to date, visual inaccuracies in their current software have often inhibited ProjectX engineers' analysis and debugging efforts.

7.4.2 Comparison of Visualization Software

ProjectX developers currently use Visual3 [20, 19] to examine and attempt to understand their solution data. The reasons are simple: Visual3 is freely available, can deal with general (linear) element types, is extendable and there is local knowledge of this software. Visual3 supports a number of usage modes; e.g., cut-planes, surface

rendering, surface contours, isosurfaces and streamlines/streaklines.[1] ProjectX developers use Visual3 to explore their solution data and to support debugging of all aspects of their solver.

At present, Visual3 is rather dated and more modern methods (e.g., those discussed in Chapter 2) exist. Visual3 is a complete, well-tested, and thoroughly-documented application with a functional GUI and a clearly-specified API. Cutting-edge visualization software often lack these features. Moreover, until recently, native high-order visualization on commodity hardware was not possible. Ultimately, ProjectX developers are not members of the visualization community; they do not have the time or expertise to dedicate towards turning prototype software and technology demonstrators into full-fledged visualization systems. As a result, ProjectX developers continue to use Visual3 since it is a robust and familiar tool.

Visual3 operates on a set of linear shape primitives (tetrahedron, hexahedron, pyramid, and prism). The solution data and computational mesh must be transferred onto these linear primitives (via interpolation) for visualization to occur. Although originally designed for continuous fields, Visual3 can handle discontinuous data (e.g., from discontinuous Galerkin solutions) by duplicating multivalued nodes. However, since ProjectX produces high-order solutions on curved meshes, some error is introduced through the linear interpolation. To decrease visualization error, each individual element of the visualization mesh can be uniformly refined[2] (in the reference space) a user-specified number of times. Since the primary use is for data exploration where areas of interest are not known beforehand, it is not possible to avoid the expense of refinement. The computational time and storage requirements grow at a rate of 8^n, where n is the refinement level. Because of this, ProjectX developers typically use zero or one level of refinement. In this paper, two levels of refinement are performed in some cases for the sake of comparison. However there is usually insufficient memory for

[1] Visual3 does not support volume rendering, so no comparisons will be made with that ElVis capability.

[2] Adaptive refinement is another option that potentially uses resources more efficiently. Such schemes are typically driven by some error criteria, but the mesh adaptation scheme used by ProjectX ensures that all elements have roughly equal data complexity. Thus, adaptive schemes were not applied here.

three levels of refinement on our typical workstations; additionally in the ever-larger simulations run by ProjectX developers and users, two levels of refinement is often infeasible as well.

7.4.3 Simulation Examples

We preface this subsection by noting that the upcoming cases are all characterized by quadratic geometry representations and cubic solution representations. These are not limitations of ProjectX nor ElVis; rather, both pieces of software can handle different order geometry and solution representations in different elements. However, they are representative of cases being examined by ProjectX engineers at present. Additionally, these relatively low polynomial orders present a "best case" for Visual3 in the comparisons to ElVis presented in this section. The differences that exist could only become more pronounced at higher polynomial orders. In the fluid dynamics cases discussed below, M_∞ denotes the Mach number, Re_c the Reynolds number with respect to chord length, and α the angle of attack.

7.4.3.1 Simulation Case 1

This case is an isolated half delta-wing geometry with a symmetry plane running down the center chord-line of the wing. The case was originally proposed [30] to demonstrate the efficacy of their adaptation strategy. Delta-wings are common geometries for computational fluid dynamics (CFD) testing due to their relatively simple geometry and the complexities involved in the vortex formation along the leading edge of the wing and the subsequent roll-up of those vortices. The equations being solved are the compressible Navier-Stokes equations. The flow conditions are $M_\infty = 0.3$, $Re_c = 4000$, and $\alpha = 12.5$. The solution was obtained through ProjectX, using an output-adaptive automated solution strategy [67].

The delta-wing geometry is linear. The computational mesh consists of 5032 linear, tetrahedral elements with 10434 total faces (linear triangles). The solution was computed with cubic polynomials.

7.4.3.2 Simulation Case 2

This case is an isolated ONERA M6 wing again with a symmetry plane running down the center chord-line. NASA designed the wing originally with intent of studying transonic flow phenomena with wind tunnel testing, and it has since become a popular CFD validation case [49]. We are presenting a subsonic, turbulent flow over the same geometry (transonic was not available). The flow conditions are $Re_c = 11.72 \times 10^6$, $M_\infty = 0.3$, and $\alpha = 3.06$. The flow is fully turbulent; the RANS equations are being solved with the Spalart-Allmaras model for closure. As before, the solution was obtained using ProjectX.

The computational mesh consists of 70,631 quadratic elements with 146,221 faces (quadratic triangles); meshing limitations restrict us to a quadratic geometry representation. The solution polynomials are cubic.

7.4.3.3 Simulation Case 3

This case is toy example meant to demonstrate the mesh-plotting capability of ElVis. The geometry is a hemisphere. The mesh is composed of 443 quadratic tetrahedra with 1,037 faces (quadratic triangles). Case 2 also uses curved elements, but the curvature is almost unnoticeable except on boundary faces. This mesh has noticeably curved elements away from the geometry as well.

The mesh is too coarse to produce any meaningful solutions. Element-wise constant data are presented because monochrome images make the geometry harder to see.

7.4.3.4 Simulation Case 4

This case is another toy example designed to show how ElVis can display negative Jacobians naturally. "Real" computational meshes with negative Jacobians are not usable; as a result these are discarded. Thus it was simpler to create a one element mesh with very obvious negative Jacobian issues. The mesh consists of one quadratic tetrahedron with four faces (quadratic triangles). The tetrahedron has corners at (0,0,0), (1,0,0), (0,1,0), and (0,0,1). The mesh was created by first placing quadratic nodes at their locations on the linear element. Then the quadratic node at (0.5,0,0)

was moved to (0.5,0,0.6), causing the element to intersect itself and leading to negative Jacobians.

7.4.4 Visualization Accuracy

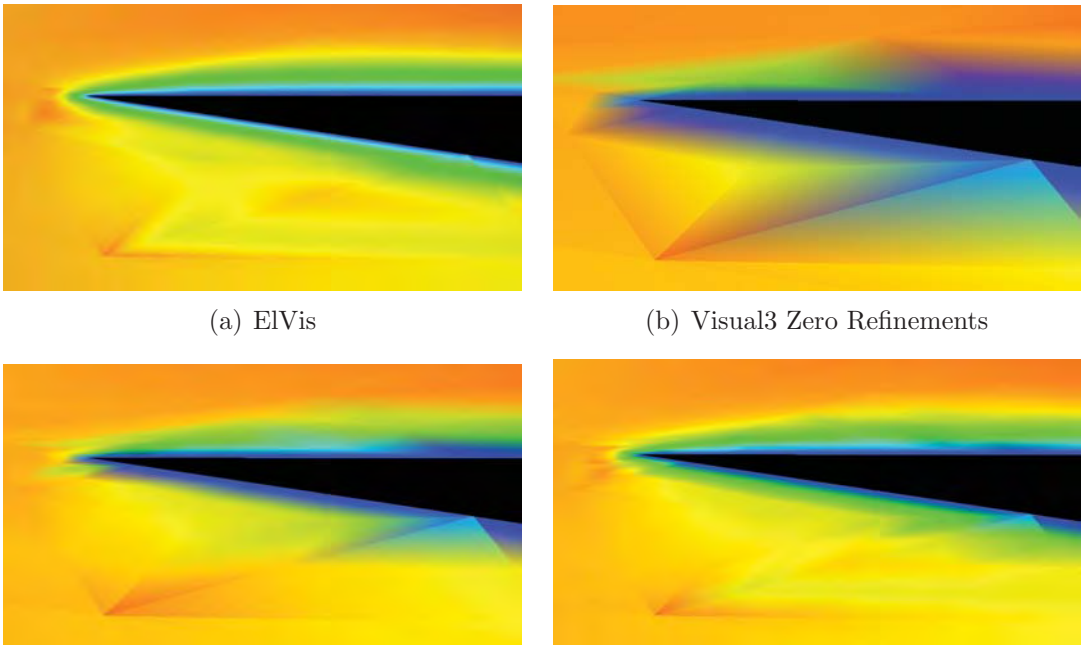
As discussed above, the primary motivation behind this work is the ability to achieve accurate visualizations of high-order data. In this section, we demonstrate ElVis' accuracy, and describe how this enables ProjectX engineers to interpret and debug their simulation data more effectively. In this subsection, we will present examples demonstrating the superior visualization accuracy possible through ElVis as compared to Visual3. Examples will be taken from surface renderings and contour lines.

7.4.5 Surface Rendering

We begin with visualizations of the leading edge of the delta wing at the symmetry plane, which we show in Figure 7.3. The black region is the airfoil at mid-span; the wing is not plotted so that it does not occlude any of the details of the boundary layer. This set of figures compares the pixel-exact rendering of ElVis to linearly-interpolated results from Visual3. For comparison, Visual3 results are posted with zero, one, and two levels of uniform refinement.

As rendered by Visual3 with zero refinements (Figure 7.3(b)), the characteristics of the solution are entirely unclear. The boundary layer appears wholly unresolved and severe mesh imprinting can be seen. It is not clear whether the apparent lack of resolution is due to a poor quality solution, bugs in the solver, or visualization errors. Even at one level of refinement (Figure 7.3(c)), the Visual3 results are still marred by visual errors. Again, mesh imprinting is substantial and the thickness of the boundary layer is not intelligible.

The Visual3 results continue to improve at two levels of refinement (Figure 7.3(d)) with the rough location of the boundary layer finally becoming apparent. But mesh imprinting is still a problem, and engineers could easily interpret this image as the result of a poor quality solution. Only the ElVis result (Figure 7.3(a)), clearly indicates the location of the boundary layer and clearly demonstrates where solution quality is locally poor due to insufficient mesh resolution. Although not shown here,



(a) ElVis (b) Visual3 Zero Refinements

(c) Visual3 One Refinement (d) Visual3 Two Refinements

Figure 7.3. Plotting mach number at the leading edge of the delta wing (Case 1) at the symmetry plane with ElVis (a) and Visual3 using zero (b), one (c), and two (d) levels of refinement.

distinct differences, similar to the effects at the leading edge on the symmetry plane, are also observed at the delta wing's trailing edge and along its entire leading edge.

ProjectX developers were genuinely surprised at the difference between between Figures 7.3(d) and 7.3(a). In fact, since users had only generated and viewed Figures 7.3(b) and 7.3(c) previously, the common misconception was that resolution at the leading edge (and indeed in many other regions around the wing) was severely lacking. Confidence in the solution accuracy arose from convergence analysis and the fact that output values matched other published results.

However, had the ProjectX solver been subject to a software bug, engineers expressed that they would have been hard pressed to interpret Visual3 results to aid these efforts. Specifically, at zero or one level of refinement, the visualization quality is so poor that developers are often unable to discern the precise source of solution artifacts. Unfortunately, a misdiagnosis can lead to a great deal of time wasted on a "wild goose chase." As a result, visualization has not played as large a

role as it could in debugging practices.

7.4.6 Contour Lines

For our next example, we illustrate the generation of contour lines. Contours are useful visualization primitives since they, unlike color plots, limit the amount of information being conveyed and allow for more detailed and targeted images. In particular, it can be difficult to interpret the magnitude and shape of a field's gradient through color maps; this is much easier with contours.

In figure 7.4 we provide an overview of the scenario and indicate the location of the cut-plane relative to the wing. In Figure 7.5, we show a comparison of contours on a cut-plane behind the trailing edge of the delta wing. The images were generated using ElVis and Visual3, once again using zero, one, and two levels of refinement for the latter.

The images in Figure 7.5 reiterate the observations from the discussion of surface rendering. With zero and one level of refinement (Figure 7.5(b) and 7.5(c)), most of the contour lines produced by Visual3 are extremely inaccurate. These images do little to illuminate the vortex structure they are trying to show. The situation improves somewhat with two levels of refinement in Visual3 (Figure 7.5(d)), but substantial errors remain. For example, the green contour running through the center of the ElVis image is one contiguous structure, whereas it is split into two regions by Visual3.

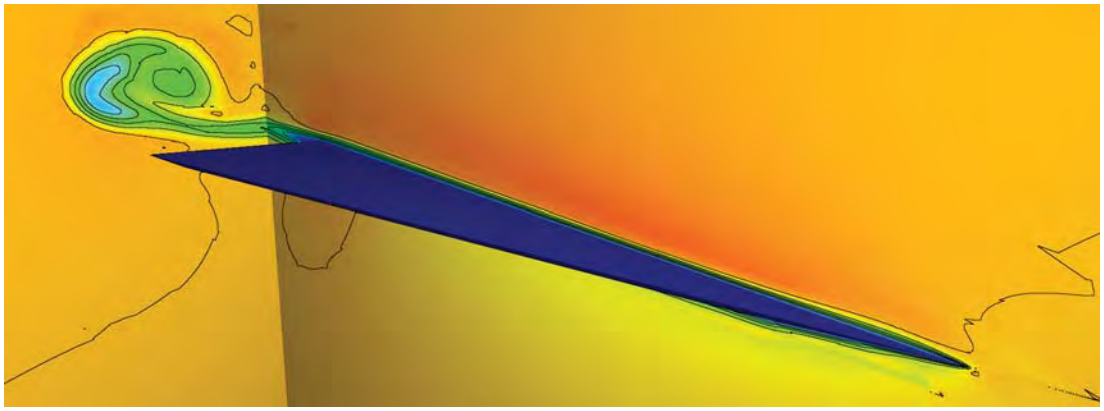
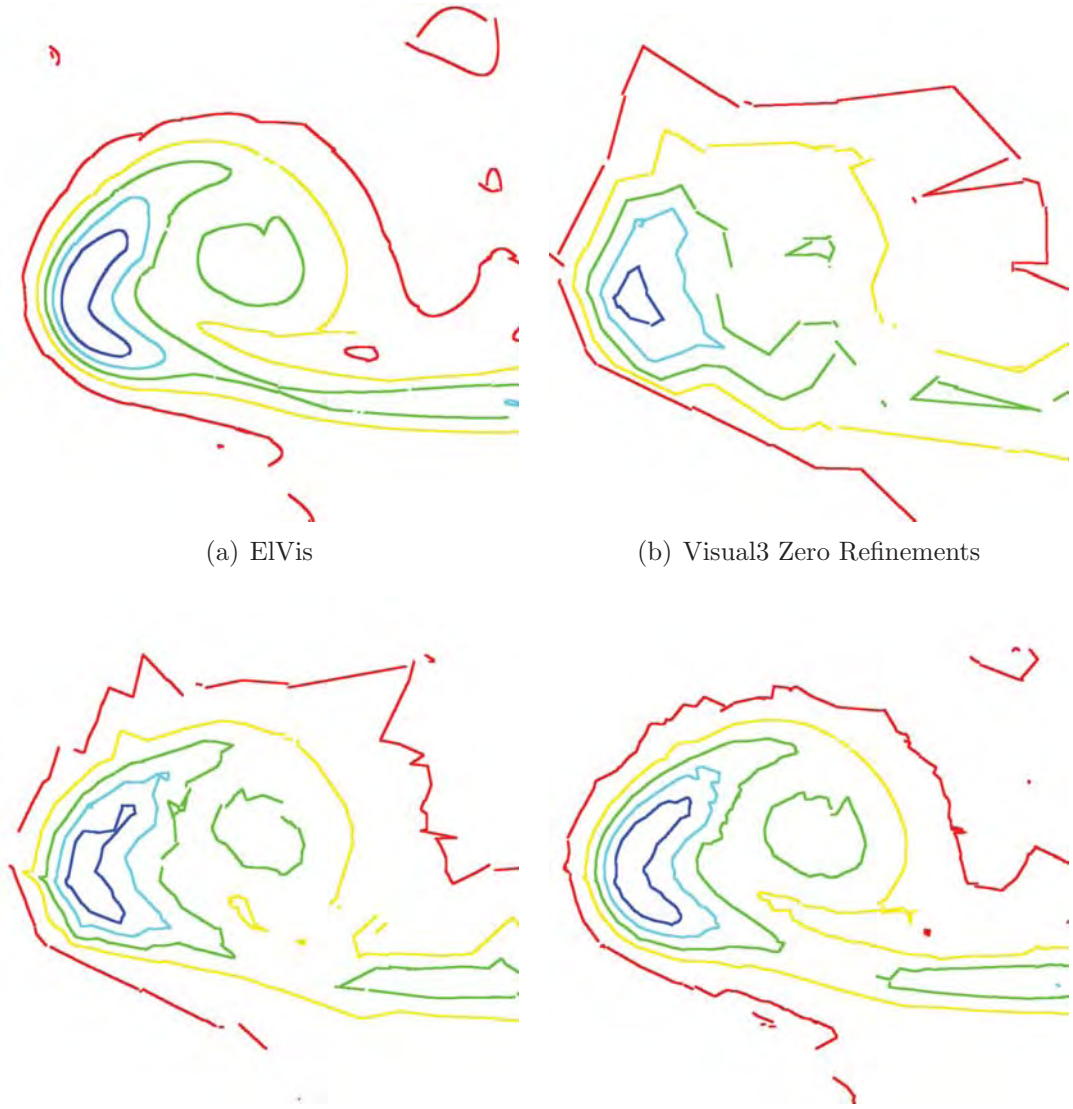


Figure 7.4. A zoomed out ElVis generated image of the delta wing from Case 1 showing the location of the cut-plane used in Figure 7.5. The cut plane is located 0.2 chords behind the trailing edge of the wing.



(a) ElVis

(b) Visual3 Zero Refinements

(c) Visual3 One Refinement

(d) Visual3 Two Refinements

Figure 7.5. Plotting mach number contours at the trailing edge of the delta wing (Case 1). An overview of this scenario is shown in Figure 7.4, with a detailed view of the contours on the trailing cut-plane generated by ElVis (a) and Visual3 using zero (b), one (c), and two (d) levels of refinement. For visual clarity, we have modified the contour lines produced by both systems so they are thicker than the default one pixel width.

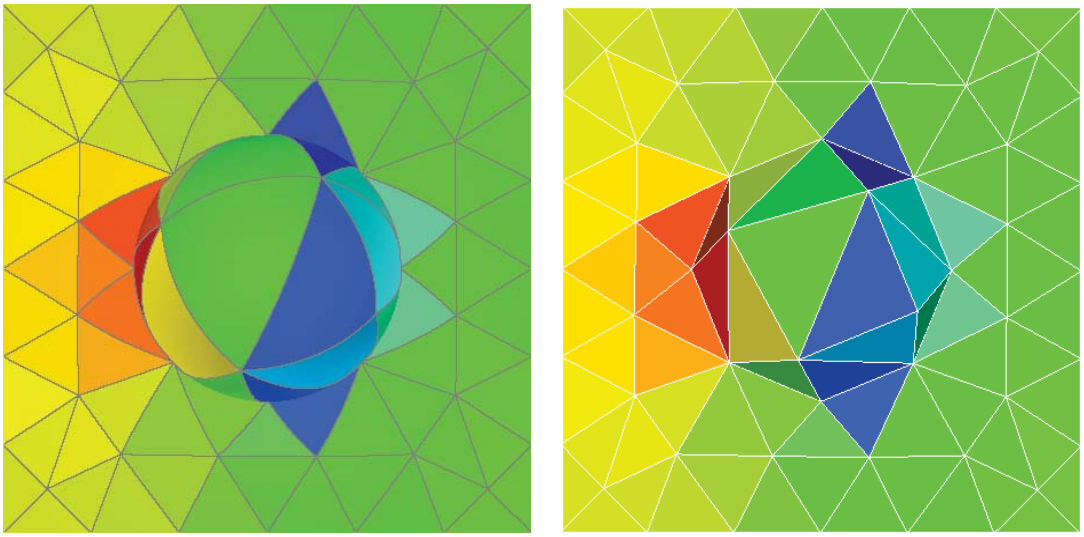
Visual3 could eventually capture these features but substantially more refinements could be required, leading to unacceptably high computational and memory usage costs.

As before with surface rendering, the errors present in the Visual3 outputs at all tested levels of refinement are too great to properly support visualization as a debugging tool. With one and even two levels of refinement, the Visual3 images are as colorful as those produced by ElVis, but nothing more. The resolution of vortex structures like the one shown in Figures 7.4 and 7.5 is a prime candidate for the application of high-order methods, because vortexes are smooth flow features. They are also extremely common, arising as important features for lift and drag calculations in any 3D lifting flow, amongst other things. Here, we are only showing a solution with cubic polynomials; with the even higher polynomial orders that could be applied to vortex flows, linear interpolation-based visualization methods will be even more inadequate.

7.4.7 Curved Mesh Visualization

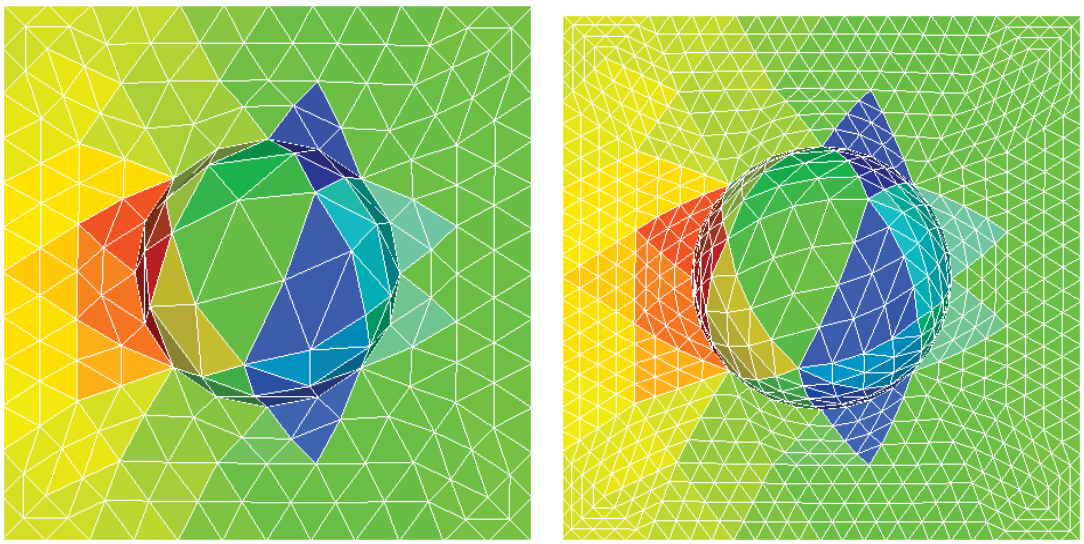
Following on the footsteps of the previous section on accuracy, it is impossible to accurately visualize curved geometries with only linear interpolation techniques. Here we will examine a mesh where highly curved elements can be seen clearly: the hemisphere from Case 3. With zero levels of refinement (Figure 7.6(b)), the Visual3 results are not obviously hemispherical at all. Figures 7.6(c) and 7.6(d), showing one and two levels of refinement respectively, provide successively greater indication that the underlying geometry is in fact curved. However, without the color scheme which helps outline true element boundaries (as opposed to boundaries generated through refinement), typical Visual3 displays can make it difficult to discern which computational element contains a particular point on the screen. This issue is further compounded by the fact that curved elements are linearized.

Figure 7.6(a) does not have any of these issues. Produced by ElVis, this representation accurately represents the curved surface. Engineers would be readily able to localize particular flow features or artifacts to specific elements for further investigation during debugging or analysis. ProjectX developers and users are given



(a) ElVis

(b) Visual3 Zero Refinements



(c) Visual3 One Refinement

(d) Visual3 Two Refinements

Figure 7.6. A top-down view of the hemisphere form Case 3. The mesh plotting tools of Visual3 and ElVis are enabled.

easy and direct access to curved geometries and curved elements, capabilities that were impossible using Visual3.

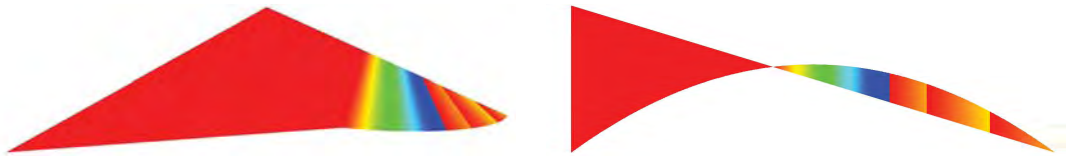
7.4.8 Negative Jacobian Visualization

Negative Jacobians can be extremely difficult to detect. In general, negative Jacobians manifest in elements whose reference to world space mapping is not invertible since it becomes multivalued. As a result, their presence can lead to severe stability and convergence issues. In general, detecting negative Jacobians amounts to a multivariate root-finding problem of high-order polynomials. This procedure is very costly and it is not practical to search every element. Instead, finite element practitioners typically check for negative Jacobians at a specific set of sample points (usually related to the integration rules used); unfortunately sampling is not a sufficient condition for detection.

If negative Jacobians are suspected (e.g., through convergence failure of the solver), visualization of problematic elements is a potential path for deciding whether Jacobian issues played a role. At present, ProjectX developers have no way of directly visualizing negative Jacobians. Linear tetrahedral elements have constant Jacobians; thus the linear visualization mesh produced for Visual3 is of little use when it is used to visualize inverted elements. However, ElVis is not subject to such constraints since it handles curved elements naturally. Case 4 demonstrates our ability to directly visualize negative Jacobians as shown in Figure 7.7. Case 4 represents a particularly severe case, but more subtle cases can still be caught visually by zooming in. That is, using visualization to verify negative Jacobians gives the user an interactive ability to increase point-sampling resolution.

Visualization of negative Jacobians also has applications in the mesh generation field. At present, metric-based 3D mesh generators[3] (the type used by ProjectX) generate curved meshes by first creating a linear mesh, and then curving it. Curving only elements with face(s) lying on the boundary is not robust since the curving can cause self-intersection (particularly with anisotropic elements). A number of heuris-

[3]Metric-based 3D mesh generation remains a challenging, open problem [38].



(a) (b)

Figure 7.7. Views of Case 4 (negative Jacobians in a single tetrahedron) rendered with ElVis. Colors show Jacobian values from -0.5 to 0.0. Left is a view of the $x + y + z = 1$ face being intersected by the $x - y$ face. Right is a view of the $x - z$ face, where the self-intersection effect of the quadratic node at (0.5,0,0.6) is apparent.

tics exist for performing this curving, but a common failure mode is the production of a mesh with negative Jacobians.

7.4.9 Distance Function Visualization

A distance function is a scalar field defined by

$$d(\vec{x}) = inf\{|\vec{x} - \vec{p}| \ : S(\vec{p}) = 0\} \tag{7.1}$$

where S is an implicit definition of a surface. In regard to ProjectX, the distance function is an important part of the Spalart-Allmaras turbulence model, which is used in Case 2. In fact, wall-distances are needed by most turbulence models. In aerospace applications, turbulent effects typically first arise due to very near-wall viscous interactions. The consequences of incorrect distance computations can vary widely, from having no effect, to producing incorrect results, to reducing solver robustness or even preventing convergence all together.

Visualization has the potential to be a valuable first attempt at diagnosing distance calculation errors. Developers can inspect the distance field for smoothness and make other qualitative judgments on the quality of the computation. Although visualization cannot guarantee that distance computations are correct, they can provide confidence and more importantly one would hope that visualization would make substantial errors in distance computations apparent.

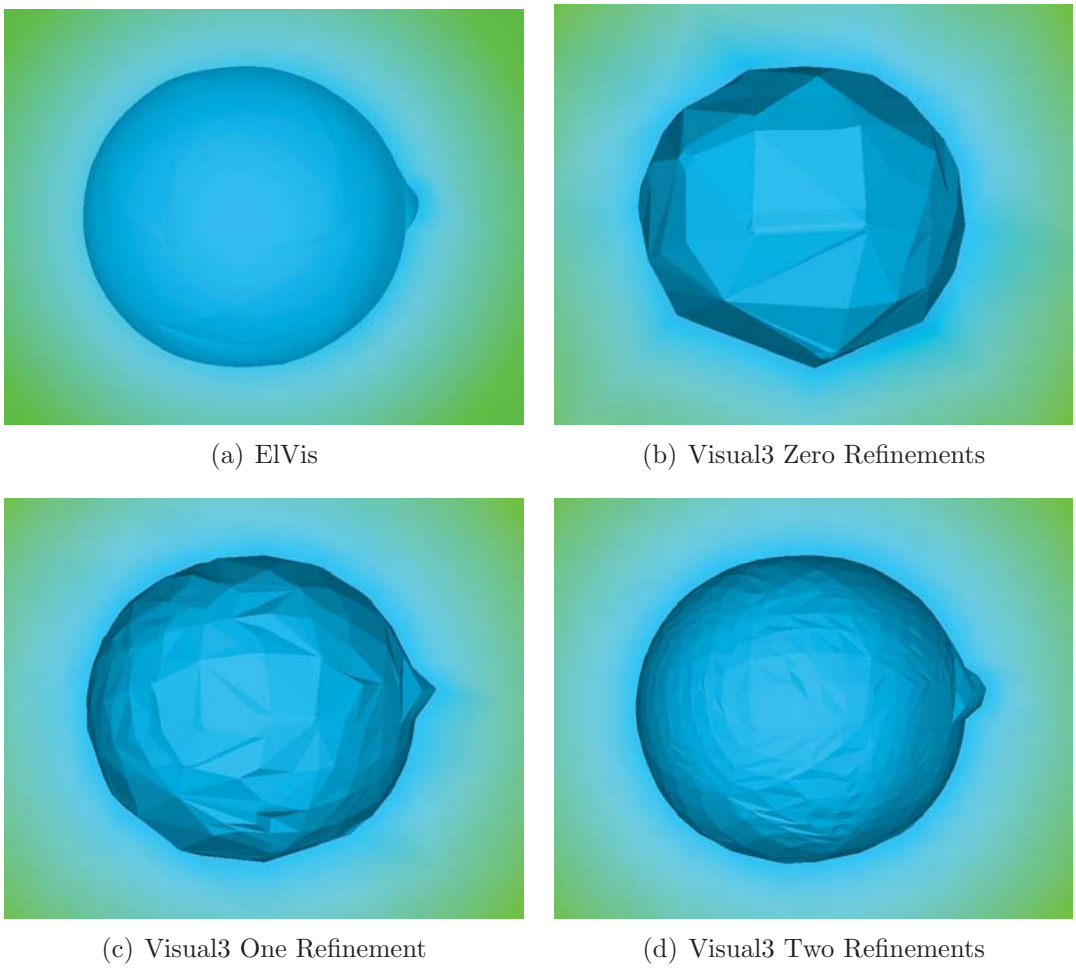
Existing visualization packages introduce error into this visualization because they must interpolate high-order surfaces and interpolate the distance data, the latter of

which is not even a polynomial field. Linear interpolation introduces a number of problems. It is impossible to judge the quality of distance calculations without at least being able to see the true shape of the underlying geometry. As a result, ProjectX developers typically find themselves unable to use visualization to aid in debugging distance computations; let us see why.

Figure 7.8 demonstrates the difficulty experienced by ProjectX developers when using Visual3 to help diagnose problems with distance function computations. In the zero refinement case (Figure 7.8(b)), the visualization mesh is so coarse that almost nothing can be learned from this image, except possibly that the distance computation is not producing completely random numbers. The 1 refinement case Figure 7.8(c)) does little to improve the situation. Here, a large protrusion is visible on the right and a large recess is visible near the top of the isosurface. Other confusing-looking regions are also present. After two levels of refinement (Figure 7.8(d)), Visual3 gives strong evidence that a bug is present in the distance computation, but the linear interpolation is still preventing rendering of the expected smooth surface.

On the other hand, ElVis renders (Figure 7.8(a)) a smooth surface with one substantial protrusion (corresponding to the protruded region seen in Figures 7.8(c) and 7.8(d)). From the ElVis result, the fact that the distance computation is wrong is obvious. Additionally, the ElVis result was obtained in seconds, in stark contrast to performing two uniform refinements in Visual3, which takes several hours.

Figure 7.9 shows the result from plotting the correct distance function in Visual3 and in ElVis. The effect of the bug was very local (manifested in the large protrusion on the right side). The images from Figure 7.8 are largely unchanged, hence only the highest resolution Visual3 image was replicated. Indeed, the zero refinement results from Visual3 with the bug fixed appears indistinguishable from Figure 7.8(b), making this level of Visual3 resolution useless for debugging. At one level of refinement, the right-side bump is gone, but there are so many other recesses and protrusions that ProjectX developers indicate they would have little confidence that the distance evaluation is correct. After two levels of refinement (Figure 7.8(d)), the Visual3 results look believable for a linear interpolation of the distance field. Nonetheless, ProjectX developers indicated that they would much rather have debugged the dis-



(a) ElVis (b) Visual3 Zero Refinements

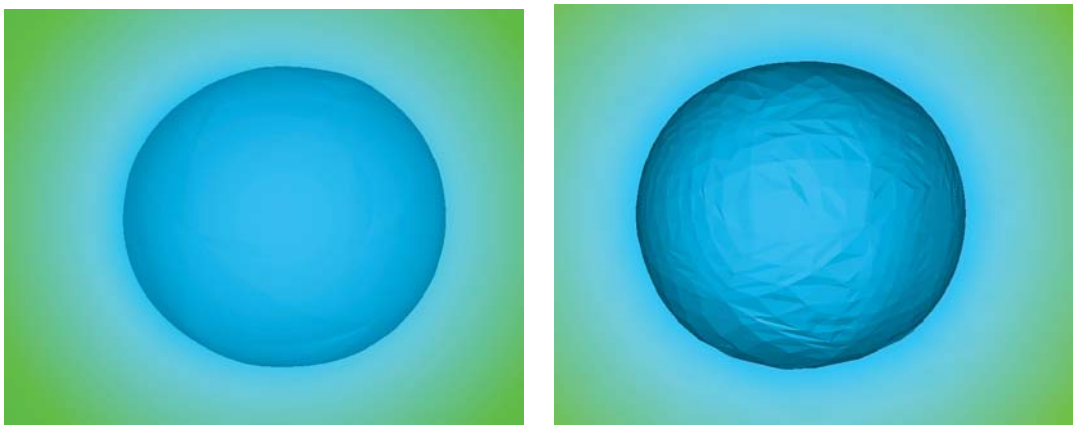
(c) Visual3 One Refinement (d) Visual3 Two Refinements

Figure 7.8. Plotting the isosurface for a distance of 6.2886 to the surface of the ONERA wing (Case 2) with ElVis (a) and Visual3 using zero (b), one (c), and two (d) levels of refinement. Here, the underlying distance computation has a bug.

tance function with ElVis, even if Visual3 could perform more refinements without additional compute and storage overhead.

In fact, ProjectX developers used the distance computations shown in Figure 7.8 for a period of months before finding the bug that caused the large protrusion shown in the isosurfaces. They had checked the distance function using views similar to those shown in Figures 7.8(b) and 7.8(c). However, due to the lack of clarity in those images and since the solver appeared to be performing reasonably, no issues were suspected.

When presented with the before and after views from Figures 7.8(a) and 7.9(a), one



(a) ElVis (b) Visual3 2 Refinements

Figure 7.9. An isosurface of the distance to the Onera wing after the bug in the underlying computation has been fixed.

ProjectX developer expressed great frustration that ElVis was not available during the development of the distance function. It would have saved him many hours of confusion, greatly accelerating the debugging process by providing clear and direct access to the underlying data.

7.5 Summary

In this chapter we presented ElVis, a new high-order finite element visualization system. ElVis was designed with an extensible architecture; it provides interactive performance; and it produces accurate, pixel-exact visualizations. The final point deserves further emphasis: ElVis makes few assumptions about the underlying data, nor does it use approximations when evaluating the solution. This degree of accuracy is a substantial step beyond what is possible in contemporary, linear interpolation-based methods. ElVis provides users with direct access to their finite element solution data either through conversion to known storage formats or by querying user-provided code directly. Direct data access is a capability previously unavailable to most finite element practitioners, and judging from the reactions of ProjectX developers, this capability is immensely valuable. We show that the software design elements behind ElVis reduce barriers to entry (in terms of coding effort) for new users. ElVis is also readily available as a full-fledged, ready-to-use application, making it a good choice

for finite element practitioners seeking a native high-order visualization system.

We have summarized and demonstrated several of ElVis' capabilities in the context of aerodynamic flow problems taken from cases previously and currently being investigated by ProjectX developers and users. These capabilities include rendering of cut-surfaces, contours and isosurfaces. Then we provided multiple comparisons between ElVis and Visual3—the visualization software presently used by ProjectX developers and users—that emphasized the shortcomings not just of Visual3, but of all linear interpolation-based visualization systems. We compared surface renderings and contouring results using data from a cubic solution, linear geometry delta wing. Even when running Visual3 at a higher level of accuracy than typically used by ProjectX developers, substantial differences were observed between Visual3 and ElVis results. The same message was repeated in Visual3's inability to accurately represent a hemisphere geometry with only moderate curvature. Finally, a potential debugging application for ElVis was presented: the evaluation of the distance field around a wing. Using Visual3 and other methods, ProjectX developers overlooked a bug in the distance calculation that persisted for months. With ElVis, the issue was immediately clear.

The capabilities demonstrated here are the components of ElVis' initial release, which has been focused on scalar field visualization. This is not a fundamental limitation of the software; future releases will address additional visualization capabilities to address additional user requests. In particular, ElVis will be enhanced to handle solutions involving cut cells [15], a capability not available in any current visualization system.

Even as it stands now, every ProjectX developer wanted to know how to "get [ElVis] on my computer" or "when can I start using [ElVis]." ElVis fills a major gap that has existed in scientific visualization. While solvers are moving toward high-order methods, visualization systems continue to apply linear interpolations. ProjectX developers and users were hard-pressed to debug their solver and analyze the results it produced. High levels of visualization errors caused developers to misdiagnose bugs and arrive at erroneous conclusions about mesh resolution, amongst other issues. These problems could have been avoided with ElVis. Ultimately, everyone involved

agrees that ElVis will be a welcome and valuable addition to their kit of development, debugging, and analysis tools.

CHAPTER 8

CONCLUSION AND FUTURE WORK

The purpose of this dissertation has been to develop new visualization algorithms for high-order spectral/*hp* scalar fields that are both accurate and interactive. The isosurface algorithm described in Chapter 4 was developed prior to the start of this dissertation. We then built upon this work by developing a new algorithm for rendering cut-surfaces through high-order fields (Chapter 5) and volume rendering (Chapter 6). In each of these cases, we demonstrated algorithms that are capable of generating pixel-exact visualizations of the underlying high-order data. We also implemented these algorithms on commodity GPUs, which enabled engineers to interactively generate these images on typical desktop systems. Working closely with the ProjectX simulation team, we developed the ElVis visualization framework (described in Chapter 7), and were able to demonstrate the effectiveness of these algorithms when applied to aerodynamic flow problems that are currently being investigated.

This dissertation's focus on high-order scalar fields is just the first step towards a truly comprehensive visualization system for high-order simulations. High-order vector fields are an important type of data that has received limited attention. While techniques exist for generating streamlines in high-order fields [61, 7], it is not immediately apparent how this will extend to a GPU implementation; additional capabilities will need to be developed for ElVis to support these features. For other types of visualizations, such as line integral convolutions on cut-surfaces and the generation of pathlines, there has yet to be any analysis on their accurate or efficient implementation for high-order fields.

Publications: The following publications relating to this work have been published:

- Blake Nelson and Robert M. Kirby. 2006. "Ray-Tracing Polymorphic Multidomain Spectral/hp Elements for Isosurface Rendering", *IEEE Transactions on Visualization and Computer Graphics* 12, 1 (January 2006), 114-125.

- Miriah Meyer, Blake Nelson, Robert M. Kirby and Ross Whitaker, "Particle Systems for Efficient and Accurate Finite Element Visualization", *IEEE Transactions on Visualization and Computer Graphics*, Vol. 13, Number 5, pages 1015-1026, 2007.

- Blake Nelson, Robert Haimes, and Robert M. Kirby. 2011. "GPU-Based Interactive Cut-Surface Extraction From High-Order Finite Element Fields", *IEEE Transactions on Visualization and Computer Graphics* 17, 12 (December 2011), 1803-1811.

The following have been submitted for publication:

- Blake Nelson, Robert M. Kirby, and Robert Haimes, "GPU-Based Interactive Volume Visualization From High-Order Finite Element Fields", IEEE Visualization Conference, 2012.

- Blake Nelson, Eric Liu, Robert M. Kirby, and Robert Haimes, "ElVis: A System for the Accurate and Interactive Visualization of High-Order Finite Element Solutions", IEEE Visualization Conference, 2012.

- Blake Nelson, Robert M. Kirby, and Steven Parker, "Optimal Expression Evaluation Through the Use of Expression Templates When Evaluating Dense Linear Algebra Operators", *ACM Transactions on Mathematical Software*.

REFERENCES

[1] ImageVis3D: A Real-time Volume Rendering Tool for Large Data. Scientific Computing and Imaging Institute (SCI).

[2] Cuda. developer.nvidia.com/category/zone/cudazone, 2012.

[3] Nektar++. http://www.nektar.info, 2012.

[4] J. Akin, W. Gray, and Q. Zhang. Colouring isoparametric contours. *Engineering Computations*, 1:36–41, 1984.

[5] M. Brasher and R. Haimes. Rendering planar cuts through quadratic and cubic finite elements. In *Proceedings of the conference on Visualization '04*, VIS '04, pages 409–416, Washington, DC, USA, 2004. IEEE Computer Society.

[6] C. Canuto, M. Hussaini, A. Quarteroni, and T. Zang. *Spectral Methods in Fluid Mechanics*. Springer-Verlag, New York, 1987.

[7] G. Coppola, S. J. Sherwin, and J. Peiró. Nonlinear particle tracking for high-order elements. *J. Comput. Phys.*, 172:356–386, September 2001.

[8] P. J. Davis and P. Rabinowitz. *Methods of Numerical Integration*. Computer Science and Applied Mathematics. Academic Press Inc., Orlando, FL, 2nd edition, 1984.

[9] M. Deville, E. Mund, and P. Fischer. *High Order Methods for Incompressible Fluid Flow*. Cambridge University Press, 2002.

[10] L. T. Diosady and D. L. Darmofal. Massively parallel solution techniques for higher-order finite-element discretizations in CFD. In *Adaptive High-Order Methods in Computational Fluid Dynamics*. World Scientific, 2011.

[11] J.-F. El Hajjar, S. Marchesin, J.-M. Dischler, and C. Mongenet. Second order pre-integrated volume rendering. In *Visualization Symposium, 2008. PacificVIS '08. IEEE Pacific*, pages 9 –16, March 2008.

[12] K. Engel, M. Kraus, and T. Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, HWWS '01, pages 9–16, New York, NY, USA, 2001. ACM.

[13] T. Etiene, L. G. Nonato, C. Scheidegger, J. Tierny, T. J. Peters, V. Pascucci, R. M. Kirby, and C. T. Silva. Topology verification for isosurface extraction. *IEEE Transactions on Visualization and Computer Graphics*, 99(RapidPosts), 2011.

[14] T. Etiene, C. Scheidegger, L. G. Nonato, R. M. Kirby, and C. Silva. Verifiable visualization for isosurface extraction. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1227–1234, Nov. 2009.

[15] K. J. Fidkowski and D. L. Darmofal. A triangular cut-cell adaptive method for higher-order discretizations of the compressible Navier-Stokes equations. 225:1653–1672, 2007.

[16] M. P. Garrity. Raytracing irregular volume data. In *Proceedings of the 1990 workshop on Volume visualization*, VVS '90, pages 35–40, New York, NY, USA, 1990. ACM.

[17] B. Haasdonk, M. Ohlberger, M. Rumpf, A. Schmidt, and K. G. Siebert. Multiresolution visualization of higher order adaptive finite element simulations. *Computing*, 70:181–204, July 2003.

[18] M. Hadwiger, J. M. Kniss, C. Rezk-salama, D. Weiskopf, and K. Engel. *Real-time Volume Graphics*. A. K. Peters, Ltd., Natick, MA, USA, 2006.

[19] R. Haimes and D. Darmofal. Visualization in computational fluid dynamics: A case study. In *IEEE Computer Society, Visualization*, pages 392–397. IEEE Computer Society Press, 1991.

[20] R. Haimes and M. Giles. Visual3: Interactive unsteady unstructured 3d visualization. AIAA 91-0794, 1991.

[21] K. Hoffman and R. Kunze. *Linear Algebra*. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1961.

[22] T. J. R. Hughes. *The Finite Element Method*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1987.

[23] G. Karniadakis, E. Bullister, and A. Patera. A spectral element method for solution of two- and three-dimensional time dependent Navier-Stokes equations. In *Finite Element Methods for Nonlinear Problems, Springer-Verlag*, page 803, 1985.

[24] G. E. Karniadakis and S. J. Sherwin. *Spectral/hp element methods for CFD*. Oxford University Press, New-York, NY, USA, 1999.

[25] R. M. Kirby and C. T. Silva. The need for verifiable visualization. *IEEE Comput. Graph. Appl.*, 28(5):78–83, Sept. 2008.

[26] J. Kniss, S. Premoze, M. Ikits, A. Lefohn, C. Hansen, and E. Praun. Gaussian transfer functions for multi-field volume visualization. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, VIS '03, pages 497–504, Washington, DC, USA, 2003. IEEE Computer Society.

[27] A. Knoll, Y. Hijazi, I. Wald, C. Hansen, and H. Hagen. Interactive ray tracing of arbitrary implicits with simd interval arithmetic. In *In Proceedings of the 2nd IEEE/EG Symposium on Interactive Ray Tracing*, pages 11–18, 2007.

[28] G. D. Kontopidis and D. E. Limbert. A predictor-corrector contouring algorithm for isoparametric 3d elements. *International Journal for Numerical Methods in Engineering*, 19(7):995–1004, 1983.

[29] A. Kronrod. *Nodes and weights of quadrature formulas, sixteen-place tables: Authorized translation from the Russian.* Consultants Bureau, 1965.

[30] T. Leicht and R. Hartmann. Error estimation and anisotropic mesh refinement for 3d laminar aerodynamic flow simulations. 229:7344–7360, 2010.

[31] A. O. Leone, R. Scateni, S. Pedinotti, L. Marzano, P. Marzano, E. Gobbetti, E. Gobbetti, and S. S. Pedinotti. Discontinuous finite element visualization.

[32] G. Marmitt, H. Friedrich, and P. Slusallek. Interactive Volume Rendering with Ray Tracing. In *Eurographics State of the Art Reports*, 2006.

[33] G. Marmitt, H. Friedrich, and P. Slusallek. Efficient cpu-based volume ray tracing techniques. *Computer Graphics Forum*, 27(6):1687–1709, 2008.

[34] N. Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, June 1995.

[35] J. L. Meek and G. Beer. Contour plotting of data using isoparametric element representation. *International Journal for Numerical Methods in Engineering*, 10(4):954–957, 1976.

[36] M. Meyer, B. Nelson, R. Kirby, and R. Whitaker. Particle systems for efficient and accurate high-order finite element visualization. *IEEE Transactions on Visualization and Computer Graphics*, 13:1015–1026, 2007.

[37] J. Meyer-Spradow, T. Ropinski, J. Mensmann, and K. H. Hinrichs. Voreen: A rapid-prototyping environment for ray-casting-based volume visualizations. *IEEE Computer Graphics and Applications (Applications Department)*, 29(6):6–13, Nov./Dec. 2009.

[38] T. Michal and J. Krakos. Anisotropic mesh adaptation through edge primitive operations. AIAA 2012-159, 2012.

[39] R. E. Moore, R. B. Kearfott, and M. J. Cloud. *Introduction to Interval Analysis.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2009.

[40] K. Moreland and E. Angel. A fast high accuracy volume renderer for unstructured data. In *Proceedings of the 2004 IEEE Symposium on Volume Visualization and Graphics*, VV '04, pages 9–16, Washington, DC, USA, 2004. IEEE Computer Society.

[41] B. Nelson, R. Haimes, and R. M. Kirby. GPU-based interactive cut-surface extraction from high-order finite element fields. *Visualization and Computer Graphics, IEEE Transactions on*, 17(12):1803 –1811, December 2011.

[42] B. Nelson and R. M. Kirby. Ray-tracing polymorphic multidomain spectral/hp elements for isosurface rendering. *IEEE Transactions on Visualization and Computer Graphics*, 12:114–125, January 2006.

[43] K. Novins and J. Arvo. Controlled precision volume integration. In *Proceedings of the 1992 workshop on Volume visualization*, VVS '92, pages 83–89, New York, NY, USA, 1992. ACM.

[44] T. Oliver and D. Darmofal. Impact of turbulence model irregularity on high-order discretizations. AIAA 2009-953, 2009.

[45] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich. Optix: A general purpose ray tracing engine. *ACM Transactions on Graphics*, August 2010.

[46] A. Patera. A spectral method for fluid dynamics: Laminar flow in a channel expansion. *J. Comp. Phys.*, 54:468, 1984.

[47] J.-F. Remacle, N. Chevaugeon, É. Marchandise, and C. Geuzaine. Efficient visualization of high-order finite elements. *International Journal for Numerical Methods in Engineering*, 69(4):750–771, 2007.

[48] C. Sadowsky, J. Cohen, and R. Taylor. Rendering tetrahedral meshes with higher-order attenuation functions for digital radiograph reconstruction. In *Visualization, 2005. VIS 05. IEEE*, pages 303 – 310, October 2005.

[49] V. Schmitt and F. Charpin. Pressure distributions on the onera-m6-wing at transonic mach numbers. AGARD AR 138, 1979.

[50] W. Schroeder, F. Bertel, M. Malaterre, D. Thompson, P. Pebay, R. O'Bara, and S. Tendulkar. Methods and framework for visualizing higher-order finite elements. *Visualization and Computer Graphics, IEEE Transactions on*, 12(4):446 –460, july-aug. 2006.

[51] W. Schroeder, F. Bertel, M. Malaterre, D. Thompson, P. Pebay, R. O'Barall, and S. Tendulkar. Framework for visualizing higher-order basis functions. In *Visualization, 2005. VIS 05. IEEE*, pages 43 – 50, 2005.

[52] W. Schroeder, K. M. Martin, and W. E. Lorensen. *The visualization toolkit (2nd ed.): an object-oriented approach to 3D graphics*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

[53] C. Singh and D. Sarkar. A simple and fast algorithm for the plotting of contours using quadrilateral meshes. *Finite Elem. Anal. Des.*, 7:217–228, December 1990.

[54] C. Singh and J. Singh. Accurate contour plotting using 6-node triangular elements in 2d. *Finite Elem. Anal. Des.*, 45:81–93, January 2009.

[55] J. M. Snyder. Interval analysis for computer graphics. *SIGGRAPH Comput. Graph.*, 26:121–130, July 1992.

[56] M. Steffen, S. Curtis, R. M. Kirby, and J. K. Ryan. Investigation of smoothness-increasing accuracy-conserving filters for improving streamline integration through discontinuous fields. *IEEE Transactions on Visualization and Computer Graphics*, 14(3):680–692, May 2008.

[57] B. Szabó and I. Babuška. *Finite Element Analysis*. John Wiley & Sons, New York, 1991.

[58] L. N. Trefethen and I. David Bau. *Numerical Linear Algebra*. SIAM, Philadelphia, PA, USA, 1997.

[59] M. Üffinger, S. Frey, and T. Ertl. Interactive high-quality visualization of higher-order finite elements. *Computer Graphics Forum*, 29(2):337–346, 2010.

[60] P. E. J. Vos, S. J. Sherwin, and R. M. Kirby. From h to p efficiently: Implementing finite and spectral/hp element methods to achieve optimal performance for low- and high-order discretisations. *J. Comput. Phys.*, 229(13):5161–5181, July 2010.

[61] D. Walfisch, J. K. Ryan, R. M. Kirby, and R. Haimes. One-sided smoothness-increasing accuracy-conserving filtering for enhanced streamline integration through discontinuous fields. *J. Sci. Comput.*, 38(2):164–184, Feb. 2009.

[62] D. F. Wiley, H. Childs, B. Hamann, and K. Joy. Ray casting curved-quadratic elements. In O. Deussen, C. D. Hansen, D. Keim, and D. Saupe, editors, *Data Visualization 2004*, pages 201–209. Eurographics/IEEE TCVG, ACM Siggraph, 2004.

[63] D. F. Wiley, H. R. Childs, B. F. Gregorski, B. Hamann, and K. I. Joy. Contouring curved quadratic elements. In *Proceedings of the symposium on Data visualisation 2003*, VISSYM '03, pages 167–176, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[64] P. L. Williams, N. L. Max, and C. M. Stein. A high accuracy volume renderer for unstructured data. *IEEE Transactions on Visualization and Computer Graphics*, 4:37–54, January 1998.

[65] J. R. Winkler. A companion matrix resultant for Bernstein polynomials. *Linear Algebra and Its Applications*, 362:153–175, 2003.

[66] J. R. Winkler. The transformation of the companion matrix resultant between the power and Bernstein polynomial bases. *Applied Numerical Mathematics*, 48:113–126, 2004.

[67] M. Yano and D. Darmofal. An optimization framework for anisotropic simplex mesh adaptation: application to aerodynamic flows. AIAA 2012–0079, Jan. 2012.

[68] M. Yano, J. M. Modisette, and D. Darmofal. The importance of mesh adaptation for higher-order discretizations of aerodynamic flows. AIAA 2011–3852, June 2011.

[69] Y. Zhou and M. Garland. Interactive point-based rendering of higher-order tetrahedral data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):2006, 2006.